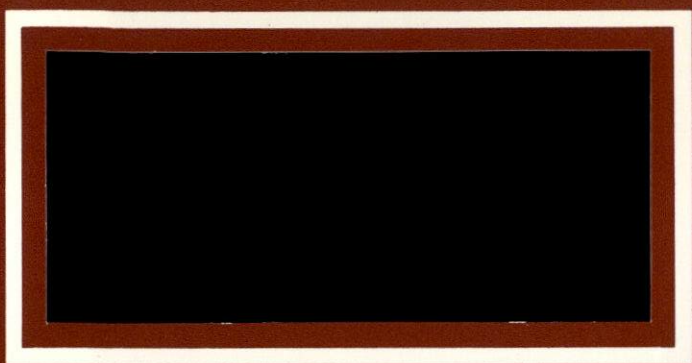




25004DSOFTWAREINC



6301 Assembler

2500AD SOFTWARE, INC.
109 Brookdale Ave, Box 480
Buena Vista, CO 81211

SALES DEPARTMENT
(719) 395-8683

TECHNICAL DEPARTMENT
(719) 395-2475

COPYRIGHT

The manual and the software described in it are copyrighted with all rights reserved. Under the copyright laws, this manual or the software may not be copied, in whole or part, without written consent of 2500AD Software, Inc., except in the normal use of the software or to make a backup copy. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold or given to another person. Under the law, copying includes translating in to another language or format.

You may use the software on any computer owned by you, but extra copies cannot be used for this purpose. For some products, a multi-use license may be purchased to allow the software to be used on more than one computer owned by the purchaser, including a shared-disk systems.

LIMITED WARRANTY ON MEDIA AND MANUALS

If you discover physical defects in the media on which this software is distributed, or in the manuals distributed with the software, 2500AD Software, Inc. will replace the media or manuals at no charge to you, provided you return the item to be replaced with proof of purchase to 2500AD Software, Inc. or an authorized 2500AD Software dealer during the 90-day period after you purchased the software. In some countries the replacement period may be different; check with your authorized dealer. **ALL IMPLIED WARRANTIES ON THE MEDIA AND MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.**

Even though 2500AD Software, Inc. has tested the software and reviewed the documentation, 2500AD SOFTWARE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS SOFTWARE, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS SOFTWARE IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE. IN NO EVENT WILL 2500AD SOFTWARE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE OR ITS DOCUMENTATION, even if advised of the possibility of such damages. In particular, 2500AD Software shall have no liability for any programs or data used with 2500AD Software products, including the costs of recovering such programs or data.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No 2500AD Software dealer, agent or employee is authorized to make any modification, extension, or addition to this warranty. Some states do not allow the exclusion of limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Table of Contents

6301 Cross Assembler

General Introduction	1 - 1
Major Version Number Differences	1 - 2
Operating Instructions	1 - 3
Prompt Mode	1 - 3
Command Line Mode	1 - 4
Input Filename Only	1 - 4
Input Filename and Output Filename	1 - 4
Listing to Terminal	1 - 5
Listing to Printer	1 - 5
Listing to Printer with Cross Reference	1 - 5
Listing to Disk	1 - 5
Listing to Disk with Cross Reference	1 - 6
Listing to Another Drive or Directory	1 - 6
Error Only Listing to Terminal	1 - 6
Error Only Listing to Printer	1 - 6
Error Only Listing to Disk	1 - 7
List On/Off to Terminal	1 - 7
List On/Off to Printer	1 - 7
List On/Off to Disk	1 - 7
System Defaults	1 - 8
Assembler Defaults	1 - 8
Linker Defaults	1 - 8
Librarian Defaults	1 - 8
Assembler Error Processing	1 - 9
Assembler Run Time Commands	1 - 10
Assembly Language Syntax	1 - 11
Number Base Designations	1 - 11
Program Comments	1 - 11
Program Counter	1 - 12
Labels	1 - 12
Local Labels	1 - 13
High Byte	1 - 14
Low Byte	1 - 14
Upper / Lower Case	1 - 14

Addressing Modes	1 - 15
Immediate	1 - 15
Direct	1 - 15
Indexed	1 - 15
Extended	1 - 16
Relative	1 - 16
Predefined Registers	1 - 17
 Assembler Directives	
Storage Control Directives	1 - 18
ORG	1 - 18
ORIGIN	1 - 18
END	1 - 18
DB	1 - 19
FCB	1 - 19
DEFB	1 - 19
BYTE	1 - 19
STRING	1 - 19
DC	1 - 19
DW	1 - 20
FDB	1 - 20
DEFW	1 - 20
WORD	1 - 20
LONG	1 - 20
LONGW	1 - 20
LWORD	1 - 20
BLKB	1 - 20
BLKW	1 - 21
BLKL	1 - 21
DS	1 - 21
RMB	1 - 21
DEFS	1 - 21
FLOAT	1 - 22
DOUBLE	1 - 22
ASCII	1 - 22
FCC	1 - 23
MASK	1 - 23
SQUOTE	1 - 23
DATE	1 - 23
Directives Specific to C Compiler Requirements	1 - 24
D_ALIGN	1 - 24
F_ALIGN	1 - 24

I_ALIGN	1 - 24
L_ALIGN	1 - 24
P_ALIGN	1 - 25
LONGCHK	1 - 25
Definition Control Directives	1 - 26
EQU	1 - 26
EQUAL	1 - 26
VAR	1 - 26
DEFL	1 - 26
SET	1 - 26
LLCHAR	1 - 26
MACRO	1 - 27
ARGCHK	1 - 27
ENDM	1 - 27
MACEND	1 - 27
MACEXIT	1 - 27
MACDELIM	1 - 28
MACFIRST	1 - 28
XDEF	1 - 28
GLOBAL	1 - 28
PUBLIC	1 - 28
GLOBALS ON	1 - 29
GLOBALS OFF	1 - 29
XREF	1 - 29
EXTERN	1 - 29
EXTERNAL	1 - 29
ASK	1 - 30
MESSAGE	1 - 30
MESSG	1 - 30
Assembly Mode Directives	1 - 31
SECTION	1 - 31
ENDS	1 - 32
ABSOLUTE	1 - 33
RELATIVE	1 - 33
RADIX	1 - 33
LEADZERO	1 - 34
INCLUDE	1 - 34
SPACES	1 - 34
TWOCHAR ON/OFF	1 - 35
COMMENT	1 - 35
ENDMOD	1 - 35
MODULE	1 - 36

Conditional Assembly Directives	1 - 37
IFZ	1 - 37
IF	1 - 37
IFNZ	1 - 37
COND	1 - 37
IFTRUE	1 - 37
IFNFALSE	1 - 37
IFNTRUE	1 - 38
IFFALSE	1 - 38
IFDEF	1 - 38
IFNDEF	1 - 38
IFSAME	1 - 39
IFNDIFF	1 - 39
IFNSAME	1 - 39
IFDIFF	1 - 39
IFEXT	1 - 40
IFNEXT	1 - 40
IFABS	1 - 40
IFNREL	1 - 40
IFREL	1 - 41
IFNABS	1 - 41
IFMA	1 - 41
IFNMA	1 - 41
ELSE	1 - 42
ENDC	1 - 42
ENDIF	1 - 42
IFCLEAR	1 - 42
EXIT	1 - 43
 Listing Control Directives	 1 - 44
LIST ON	1 - 44
LIST	1 - 44
LIST OFF	1 - 44
NOLIST	1 - 44
NLIST	1 - 44
MACLIST ON	1 - 44
MLIST	1 - 44
MACLIST OFF	1 - 45
MNLIST	1 - 45
CONDLIST ON	1 - 45
CONDLIST OFF	1 - 45
ASCLIST ON	1 - 45
ASCLIST OFF	1 - 46
PW	1 - 46
PL	1 - 46

TOP	1 - 46
PASS1 ON	1 - 47
PASS1 OFF	1 - 47
PAG	1 - 47
PAGE	1 - 47
EJECT	1 - 47
NAM	1 - 48
STRING	1 - 48
TTL	1 - 48
TITLE	1 - 48
HEADING	1 - 48
STTL	1 - 48
SUBTTL	1 - 48
SUBTITLE	1 - 48
Linker Control Directives	1 - 49
FILLCHAR	1 - 49
RECSIZE	1 - 49
SYMBOLS	1 - 49
OPTIONS	1 - 50
LINKLIST	1 - 50
COMREC	1 - 50
Assembly Time Operations	1 - 51
Calculations	1 - 51
Comparisons	1 - 52
PAGE 0	1 - 53
Absolute Versus Relative	1 - 54
Macros	1 - 56
Definition	1 - 56
Argument Separators	1 - 56
Labels	1 - 57
String Concatenation	1 - 57
Value Concatenation	1 - 58
Mnemonic Definitions	1 - 59
Macro Examples	1 - 60
MACFIRST	1 - 62
MACDELIM	1 - 62
Recursion	1 - 63
MACEXIT	1 - 63
Assembler Error Messages	1- 64

2500 A.D. Linker - Version 4.05

Introduction	2 - 1
Operating Instructions	2 - 2
Prompt Mode	2 - 2
Input	2 - 2
Output	2 - 2
Data File Mode	2 - 3
Command Line Mode	2 - 4
Linker Options	2 - 5
Address Relocation	2 - 6
Linker Examples	2 - 7
Single File Assembled At Desired Run Address	2 - 7
Single File With Multiple Sections	2 - 9
Multiple Files With Multiple Sections	2 - 11
Single File With One Reference-Only Section	2 - 13
Indirect Linking	2 - 15
Linker Output	2 - 20
Symbol Tables	2 - 20
Global Symbol Table	2 - 20
Abbreviated Global Symbol Table	2 - 21
MicroTek Symbol Table	2 - 22
Extended MicroTek Symbol Table	2 - 23
Zax Symbol Table	2 - 24
File Formats	2 - 25
Intel Hex Format	2 - 25
Extended Intel Hex Format	2 - 26
Motorola S19 Format	2 - 27
Motorola S28 Format	2 - 28
Motorola S37 Format	2 - 29

2500 A.D. Librarian - Version 4.05

General Description	3 - 1
Storage and Memory	3 - 1
Commands	3 - 1
System Defaults	3 - 1
Screen	3 - 2
Modules	3 - 2
Installation	3 - 3

Operating Instructions	3 - 4
Command Line Mode:	3 - 4
Prompt Mode:	3 - 4
Librarian Commands	3 - 5
ADD	3 - 5
A	3 - 5
DEL	3 - 6
D	3 - 6
EXIT	3 - 6
FIND	3 - 7
HELP	3 - 7
H	3 - 7
LIST	3 - 8
NEW	3 - 8
N	3 - 8
QUIT	3 - 9
STAT	3 - 9
S	3 - 9
REP	3 - 10
R	3 - 10
TOP	3 - 11
BOT	3 - 11
Librarian Error Messages	3 - 12
Appendix A	
Ascii Tables	A - 1
Ascii Chart	A - 1
Abbreviations for Ascii Control Characters	A - 5
Appendix B	
System Requirements	A - 6
Index	I - 1

ASSEMBLER

ASSEMBLER

6301 Cross Assembler

General Introduction

This Introduction gives a general description of what the 2500 A.D. 6301 Cross Assembler can do. We assume that you are already familiar with the 6301's operation and instruction set.

With this assembler you can write programs and assemble them into relocatable object code. You can then use the Linker to link them to the execution addresses you want.

The assembler will process files of any size so long as you have enough memory. It requests and expands all buffers as it needs them, except the Source Code Input, Object Code Output, and Listing buffers, which automatically overflow to disk.

Conditional Assembly lets you assemble sections of a source file depending on the outcome of assembly-time operations. You can nest conditionals to 248 levels. To help you detect nesting errors, the assembler checks for unbalanced conditional levels and also displays the level of the current active condition in the listing's object code field.

The assembler does **Assembly Time Calculations** in 80-bit arithmetic with up to 16 pending operands. You can tailor the algebraic hierarchy with parentheses.

Listing Control lets you list an entire program or individual sections. Its error-detection display overrides listing destinations to make sure that you always know what and where the errors are.. You can use **Run Time Commands** to change listing mode on the fly. With the **LINKLIST** command you can relocate listings.

The **Linker** lets you link files or use them as external references. It too requests and expands its buffers as needed. It can output several formats, which you can change by selecting a Linker option or with an assembler **Directive**. You can define your own section names. The Linker can process up to 256 of them.

With the **Librarian** you can create a library of object modules. These you can link to any program you create with the assembler. You can therefore store useful routines for reuse any time you need them.

Major Version Number Differences

The Assembler, Linker and Librarian will always have the same major version number.

- 4.00** The first Series 4.0 Assembler release. A completely rewritten product compatible with the 3.0 Assembler series but with increased speed and added features. The Linker was very different. It could link files from any 2500 A.D Series 4.0 Assembler and produce all popular output formats and symbol table formats.
- 4.01** The Librarian was added. Refer to the **Librarian** section of this manual for details. **Files assembled with a 4.01 Assembler must be linked with a 4.01 Linker.**
- 4.02** The **LINKLIST** directive was added, enabling the Linker to modify Assembler listings to reflect the actual run-time addresses. Refer to the **LINKLIST** directive in this manual for details. **Files Assembled with a 4.02 Assembler must be linked with a 4.02 Linker.**
- 4.03** Linker **SECTION** controls were added, making it possible to control linking of any or all sections directly in the source file. Refer to the **SECTION** directive in this manual for details. **Files Assembled with a 4.03 Assembler can be linked with a 4.02 Linker, but section linking information will be ignored.**
- 4.04** Several internal modifications were made to maintain compatibility with the 2500 A.D. C Compiler, Version 4.04
- 4.05** A change of version number only, to keep it inline with the 4.05 version of the 2500 A.D. C Compiler.

Operating Instructions

Prompt Mode

To run the assembler, type : **x6301** You will see the prompt:

Listing Destination ? (N, T, P, D, E, L, <CR> = N) :

N	=	None
T	=	Terminal
P	=	Printer (Single User Systems Only)
D	=	Disk
E	=	Error Only
L	=	List On/Off

The assembler next prompts for the source code filename:
Input Filename

A source filename extension is optional. If you don't specify one, the assembler looks for the default extension **.asm**.

Once you have entered the input filename the assembler prompts:
Output Filename:

If you enter just a carriage return, the output file will have the same filename as the input file, with the extension **.obj**. If you specify a filename but not an extension, the output will have that filename with the default extension **.obj**.

If you are controlling listing with **List On/Off** directive, there will be an additional prompt:

LIST ON/OFF Listing Destination (T, P, D, <CR> = T) :

List On/Off lets you list selected parts of the source file. For details, see **Listing Control**.

If you select **Error Only**, the assembler prompts for a destination:
Error Only Listing Destination (T, P, D, <CR> = T) :

If you select listing to printer (available on single user systems only) or to disk, the assembler prompts for a Cross Reference listing.

Command Line Mode

You can also invoke the assembler with a command line. In it you must specify the input filename, and can also specify the filename and destination of output. Here is the general form of the command, with optional fields in brackets:

x6301 [-q] input_filename [output_filename] [-t, -p, -d, -px, -dx]

The **-q** (Quiet mode) option precedes the input filename. In Quiet mode the assembler outputs nothing to screen except error messages and their line numbers. This option must precede the input filename.

Here are specific command examples

Input Filename Only

x6301 input_filename

Tells the Assembler to process the source file **input_filename**.

The assembler assumes that:

- a) filename extension, since not specified, is **.asm**
- b) listing options, since not specified, default to **Error Only**, with the screen as destination.
- c) output filename, since not specified, defaults to the input filename with extension **.obj**.

Input Filename and Output Filename

x6301 input_filename output_filename

Tells the assembler to name the object file **output_filename**.

Listing to Terminal

x6301 input_filename output_filename -t

Assemble input file **input_filename.asm**, generate object file **output_filename.obj**, and send the listing to the terminal.

Listing to Printer

x6301 input_filename -p

Assemble input file **input_filename.asm** and send the listing to the printer. Output filename defaults to **input_filename.obj**. The printer option is available only on single-user systems.

Listing to Printer with Cross Reference

x6301 input_filename output_filename -px

Assemble **input_filename.asm**, generate object file **output_filename.out**, and send the listing and a cross-reference table to the printer. The printer option is available only on single-user systems

Listing to Disk

x6301 input_filename output_filename -d

Assemble **input_filename.asm**, generate **output_filename.obj**, and send the listing to disk. The disk file will also be called **output_filename**, but its extension will be **.lst**.

Listing to Disk with Cross Reference

x6301 input_filename output_filename -dx

Assemble **input_filename.asm**, generate **output_filename.obj**, and send the listing and a cross reference table to disk.

Listing to Another Drive or Directory

MSDOS

x6301 input_filename output_filename -d,a:

x6301 input_filename output_filename -d, \new

UNIX

x6301 input_filename output_filename -d, /new/

Sends the listing to a drive/directory other than the current one.

Error Only Listing to Terminal

x6301 input_filename output_filename -et

Assemble **input_filename.asm**, create **output_filename.obj**, and send error messages to the terminal.

Error Only Listing to Printer

x6301 input_filename output_filename -ep

Assemble **input_filename.asm**, create **output_filename.obj**, and send error messages to the printer. The printer option is available only on single-user systems.

Error Only Listing to Disk

x6301 input_filename output_filename -ed

Assemble **input_filename.asm**, create **output_filename.obj**, and send error messages to printer. The disk file will also be **output_filename**, but its extension will be **.lst**.

List On/Off to Terminal

x6301 input_filename -lt

Assemble **input_filename.asm**, create **output_filename.obj**, and send **LIST ON/OFF** blocks to terminal.

List On/Off to Printer

x6301 input_filename -lp

Assemble **input_filename.asm**, create **output_filename.obj**, and send **LIST ON/OFF** blocks to printer. This option is only available on single-user systems.

List On/Off to Disk

x6301 input_filename -ld

Assemble **input_filename.asm**, create **output_filename.obj**, and send **LIST ON/OFF** blocks to disk. The disk file will also be called **output_filename**, but its extension will be **.lst**.

System Defaults

If you do not specify a filename extension, 2500 A.D. programs will supply the following defaults:

Assembler Defaults

asm	-	Input to the Assembler
obj	-	Output from the Assembler
pak	-	Packed output from the Assembler
lst	-	Listing file

Linker Defaults

obj	-	Input to the Linker
lib	-	Library file
tsk	-	Executable Object Code
hex	-	Intel Hex and Extended Intel Hex
tek	-	Tektronix Hex
s19	-	Motorola S19
s28	-	Motorola S28
s37	-	Motorola S37

Librarian Defaults

obj	-	Input to the Librarian
pak	-	Packed input to the Librarian
lib	-	Output from the Librarian

Note: The assembler's output files always include additional information for the Linker. To remove this and to generate an output file with the format you want, you must always run the Linker. This is important even if your program is assembled at the correct run address and there are no external references.

Assembler Error Processing

When the assembler finds an assembly error, its behavior depends on the current listing mode.

In No List mode it will output an error message and the statement causing the error, display this information on screen, and then halt, just as if you had typed ^S. This lets you see exactly where the error is. This will happen on pass 1 and pass 2. (Some errors, such as undefined symbols, are not detectable on pass 1.) After the error display you can turn off output with ^N.

When listing to printer or disk, the assembler does not halt. It sends errors on pass 1 to terminal but not to printer/disk. On pass 2, error output goes to printer/disk as well as to terminal, and assembly continues.

Assembler Run Time Commands

You can use these commands during the assembly process, on both pass 1 and pass 2. They override on the fly the listing mode you specified when first invoking the Assembler.

UNIX

Ctrl S	-	Stop terminal output
Ctrl Q	-	Start terminal output
Del C	-	Terminate the assembly
Del T	-	Display the output at the terminal
Del D	-	Send the output to the disk
Del M	-	Multiple output (Terminal & Disk)
Del N	-	No output

MSDOS

Ctrl S	-	Stop terminal output
Ctrl Q	-	Start terminal output
Esc C	-	Terminate the assembly
Esc T	-	Display the output at the terminal
Esc P	-	Display the output at the printer
Esc D	-	Send the output to the disk
Esc M	-	Multiple output (Terminal & Disk)
Esc N	-	No output

Assembly Language Syntax

Number Base Designations

Number bases are specified by the following:

Binary	-	B
Octal	-	O or Q
Decimal	-	D or no base designation
Hex	-	H or preceding \$ sign
Ascii	-	Single or double quotes - "X" or 'X'

The two character sequences between single or double quotes shown below are predefined. However, the **TWOCHAR ON** directive must be used to enable these.

"CR"	or	'CR'	-	Carriage return
"LF"	or	'LF'	-	Line feed
"SP"	or	'SP'	-	Space
"HT"	or	'HT'	-	Horizontal tab
"NL"	or	'NL'	-	Null

Program Comments

Comment lines must start with a semi-colon or asterisk in column 1, unless you use the **COMMENT** directive. In Spaces Off mode, comments after an instruction do not need a semi-colon if there is at least 1 space or tab before the comment. In Spaces On mode, all comments after an instruction must be preceded by a semi-colon. For details and defaults, see **SPACES** directive.

Program Counter

To specify the program counter in an expression you can use the special characters dollar sign (\$) or asterisk (*). The value assigned to \$ or * is the program counter value at the start of the instruction.

Labels

Non-local labels can have any number of characters, but only 32 are significant. A label can start in any column if its name ends with a colon. If there is no colon the label must start in column 1. All labels must start with an alpha character. Label names are case-sensitive.

Local Labels

A Local Labels are used much the same as non-local labels; but the definition of a local label is only valid between non-local labels. A "local" area is one bounded by labels which keep their definition throughout the entire program. When a program passes from one local area to the next, you can reuse local label names. For since a local label is referenced only within its own "local area", a new label name is not necessary. Here are examples of the use of Local Labels:

LABEL1:	OR	LABEL1:
?1: NOP		1?: NOP
?2: JMP ?1		2?: JMP 1?
JMP ?2		JMP 2?

LABEL2:	LABEL2:
?1: NOP	1? NOP
?2: JMP ?1	2? JMP 1?
JMP ?2	JMP 2?

LABEL3:	LABEL3:
?1: NOP	1? NOP
?2: JMP ?1	2? JMP 1?
JMP ?2	JMP 2?

There are three non-local labels, LABEL1, LABEL2 and LABEL3. Local Labels, ?1 and ?2 (or 1? and 2?) have different definitions when referenced in different local areas. 1? is not the same as ?1. You can use any character, and up to 32 of them, in a local label. Never use operators like + in Local Labels. If you use directives **VAR**, **DEFL**, **SECTION**, **ENDS** and **\$**, local labels will not terminate. **EXTERN/EXTERNAL** will terminate the scope of a local label.

The assembler normally identifies a local label by the (?) prefix or suffix. You can change this identifier with the **LLCHAR** directive (See **Directives, Definition Control**).

High Byte

To load the high byte of a 16 bit value, use `>`, the unary greater than sign. This lets you use bits 8 through 15 as a relocatable byte value.

Low Byte

To load the low byte of a 16 bit value, use `<`, the unary less than sign. This lets you use bits 0 through 7 as a relocatable byte value.

Upper / Lower Case

A letter in upper case is different from the same letter in lower case. Thus all labels, including macro and section names, are case-sensitive.

Addressing Modes

Immediate

The data is contained in the instruction.

LDA	A,#12H	; Load A with the value 12h
LDAB	#DATA	; Load B with the value associated ;with DATA

Direct

The 8-bit address of the operand is contained in the instruction.

LDAA	12H	;Load A with the contents of memory ;location 12h
LDA	B,ADDRESS	;Load B with the contents of the memory ;location of the label ADDRESS

Indexed

The operand address is the sum of the 8 bit offset in the instruction and the contents of X.

LDAB	12H,X	;Ld B with contents of the memory ; location pointed to by adding 12 to ; the contents of register X
STAA	ADDRESS,X	;Store A in the memory location ; obtained by adding the value ; associated with ADDRESS ;to the contents of register X.

Extended

The 16-bit operand address is contained in the instruction.

STX	\$1234	; Store X in the memory location ; 1234h
INC	2345H	; Increment the value of memory ; location 2345h

Relative

The operand address is relative to the current instruction. If the address is given using a numerical value, the calculation is from the start of the next instruction.

BRA	LOOP	;The Assembler calculates the address ; by subtracting the address of the of ;the next instruction from the address ; of LOOP
BRA	4	;The destination is 4 bytes past the ; start of the next instruction

Predefined Registers

Name	Address	Description
P1DDR	00	Port 1 Data Direction Register
P2DDR	01	Port 2 Data Direction Register
PORT1	02	Port 1 Data Register
PORT2	03	Port 2 Data Register
P3DDR	04	Port 3 Data Direction Register
P4DDR	05	Port 4 Data Direction Register
PORT3	06	Port 3 Data Register
PORT4	07	Port 4 Data Register
TCSR	08	Timer Control and Status Register
TIMER		Timer/Counter Register
TCRH	09	T/C (MSB)
TCRL	0A	T/C (LSB)
OUTCMP		Output Compare Register
OCRH	0B	O C R (MSB)
OCRL	0C	O C R (LSB)
INCAP		Input Capture Register
ICRH	0D	I C R (MSB)
ICRL	0E	I C R (LSB)
P3CSR	0F	Port 3 Control and Status Register
RMCR	10	SCI Rate and Mode Control Register
TRCSR	11	Transmit/Receive Control/ Status Register
SCIRDR	12	SCI Receiver Data Register
SCITDR	13	SCO Transmit Data Register
RAMCR	14	RAM Control Register
	15 - 1F	Reserved bytes

Assembler Directives

A period may precede any directive to distinguish it from a program instruction.

Storage Control Directives

ORG	VALUE
ORIGIN	

Sets the program assembly address. If this directive is not executed, the assembly address defaults to 0000.

END	VALUE
-----	-------

Defines the end of a program or an included file. VALUE (optional) specifies the program starting address, which is encoded in the output file if a program starting address record type exists in the output format definition.

LABEL:	DB FCB DEFB BYTE STRING	VALUE
--------	-------------------------------------	-------

Stores **VALUE** in consecutive memory locations. **VALUE** (optional) may be any mix of operand types, each separated by a comma. You must bracket Ascii character strings with apostrophes, using two apostrophes to indicate an embedded apostrophe. If you omit **VALUE**, **BYTE** reserves and zeroes one byte. A label is optional.

.BYTE		;Reserves 1 zeroed byte.
.BYTE	10	;Reserves 1 byte = 10 decimal.
.BYTE	1,2,3	;Reserves 3 bytes, = 1,2, 3 in that order.
.BYTE	SYMBOL-10	;Searches the symbol table for ;SYMBOL, subtracts 10 decimal from ;its value, and stores the result.
.BYTE	'Hello'	;Stores the Ascii equivalent of the string ; Hello in consecutive memory locations.
.BYTE	'Hello', 0DH	;Same as before, with a carriage ; return at the end. Spaces before ; operands are ignored but the comma ; is required.
.BYTE	'2500 A.D.'s'	;Embedded apostrophe.

LABEL:	DC	VALUE
--------	----	-------

DC works like **BYTE**, storing **VALUE** in consecutive memory locations. However, it also sets the high bit of the last data byte. **DC** is useful for storing strings, since the setting of bit 7 flags the end of the string.

LABEL: DW VALUE
FDB
DEFW
WORD

Stores **VALUE** (optional) in a 16-bit storage location. Initialize multiple words by separating each expression with a comma. If you omit **VALUE**, **WORD** reserves and zeroes 1 word. A label is optional.

LABEL: LONG VALUE
LONGW
LWORD

Stores **VALUE** in a 32-bit storage location. Initialize multiple long words by separating each expression with a comma. If you omit **VALUE**, **LONG** reserves and zeroes 1 long word. A label is optional.

LABEL: BLKB SIZE, VALUE

Reserves the number of bytes specified by **SIZE** and stores **VALUE** (optional) in each byte. If you omit **VALUE**, **BLKB** zeroes the reserved bytes. A label is optional.

BLKB 20 ;Reserves 20 zeroed bytes
BLKB 20,0 ;Reserves 20 zeroed bytes
BLKB 20,FFH ;Reserves 20 bytes and stores FFh in each

LABEL: BLKW SIZE, VALUE

Reserves the number of 16 bit words specified by **SIZE** and stores **VALUE** (optional) in each word. If you omit **VALUE**, **BLKW** zeroes the reserved words. A label is optional.

BLKW	20	;Reserves 20 zeroed words
BLKW	20,0	;Reserves 20 zeroed words
BLKW	20,FFFFH	;Reserves 20 words and stores FFFFh in ;each

LABEL: BLKL SIZE, VALUE

Reserves the number of 32 bit long words specified by **SIZE** and stores **VALUE** (optional) in each long word. If you omit **VALUE**, **BLKL** zeroes the reserved long words. A label is optional.

BLKL	20	;Reserves 20 zeroed long words
BLKL	20,0	;Reserves 20 zeroed long words
BLKL	20,FFFFH	;Reserves 20 long words and stores FFFFh ;in each

LABEL: DS SIZE
RMB
DEFS

Reserves the number of bytes specified by **SIZE**, but stores nothing in the reserved area. If the storage locations are at the end of a program section and the linker output is executable and you haven't told the linker to stack another module on top of this section, then **DS** does not include the reserved bytes in the output file..

LABEL:	FLOAT	VALUE
--------	-------	-------

Converts **VALUE** to single-precision floating-point format. If the mantissa is larger than 24 bits, **FLOAT** truncates the value. You may not use scientific notation with **FLOAT**.

```

FLOAT    178.125
FLOAT    100.,125,-178.125.

```

LABEL: DOUBLE VALUE

Converts **VALUE** to double-precision floating-point format. If the mantissa is larger than 52 bits, **DOUBLE** truncates the value. You may not use scientific notation with **DOUBLE**.

DOUBLE 178.125
DOUBLE 100.,125,-178.125

LABEL: ASCII STRING

Stores **STRING** in memory up to but not including either a carriage return or a broken bar character ("|", Hex 7C). A label is optional.

ASCII **Hello** ;Stores the Ascii representation of Hello (and this
;comment) in consecutive memory locations.

ASCII **Hello|** ;Now the comment is not stored.

ASCII **Hello<cr>** ;Termination with a carriage return only

LABEL: FCC STRING

Stores **STRING** in memory until a character is reached that matches the first character. The first character and the second matching character are not stored. A label is optional.

FCC /This is a test string/ ;does not store the slashes

MASK VALUE, VALUE

Defines a mask for ascii character literals and strings. The ascii characters are logically and-ed with the first value and or-ed with the second value. The default values are FFh and 00h respectively.

SQUOTE

Tells the assembler to treat an ascii string with only a leading apostrophe as if it were bracketed by a pair of apostrophes.

LABEL: DATE

Stores the current date in ascii.

Under UNIX the format is: day of the week (3 characters), month (3 characters), date (dd), time (hh:mm:ss), and year (yyyy). The field delimiter is a single space. The label is optional.

On MSDOS the format is: day of the week (3 characters), month (3 characters), date (dd), year(yyyy), and time (hh:mm)

Directives Specific to C Compiler Requirements

The following directives do not relate directly to the function of the Assembler, but are necessary with some hardware to meet C alignment requirements and thus avoid run-time errors in compiled C programs.

D_ALIGN

Aligns the program counter with the next double floating point number boundary.

F_ALIGN

Aligns the program counter with the next floating point number boundary.

I_ALIGN

Aligns the program counter with the next integer number boundary.

L_ALIGN

Aligns the program counter with the next long number boundary.

P_ALIGN

Aligns the program counter with the next pointer boundary.

LONGCHK ON/OFF

When this switch is **OFF** the assembler does not do its normal overflow error-check on 32-bit numbers, but simply accepts them. This is primarily of concern in using the C compiler, which does not consider 32-bit integer overflows to be errors.

When **LONGCHK** is **ON** error-checking occurs. This is the default setting.

Definition Control Directives

LABEL: EQU VALUE
EQUAL

Equates LABEL to VALUE. VALUE may be another symbol or any legal arithmetic expression. Do not forward-reference VALUE, as this would produce the error **LABEL VALUE changed between passes.**

LABEL: VAR VALUE
DEFL
SET

Equates LABEL to VALUE. You can change the assignment as often as you like throughout the program. You should not use VAR to redefine a label defined as a variable.

LLCHAR CHARACTER

Changes the character that identifies a Local Label. The default identifier is ?. You should avoid the characters that designate number bases, but if you have to use them, they must appear only at the end of the label.

LABEL: MACRO ARGS

Specifies the start of a Macro Definition.

ARGCHK ON/OFF

Use **ARGCHK OFF** to invoke a macro with fewer parameters than were used in defining it. **ARGCHK OFF** turns off parameter checking. The default is **ON**.

**ENDM
MACEND**

Specifies the end of a Macro Definition.

MACEXIT

Exits from a macro immediately and unconditionally. Unlike **MACEND**, **MACEXIT** does not terminate a macro definition. It simply exits from wherever it is placed within a macro, leaving the rest unexecuted and restoring all conditional values to what they were before the macro was invoked.

MACDELIM CHARACTER

Allows you to pass an argument containing a comma into a macro. Normally commas default to being argument separators. **MACDELIM** changes the delimiter character, letting you embed commas in a string. {, (and [are the only valid substitute characters.

MACFIRST

This directive switches the default order of Assembler table searches, giving search priority to the Macro Definition Table over the Mnemonic Table. (See "Mnemonic Definitions" under **Macros**.)

LABEL: **XDEF**
 GLOBAL
 PUBLIC

Defines **LABEL** as a global label that other programs can reference. Define multiple labels by separating each with a comma.

GLOBAL SYM1 ; Declares the label SYM1 accessible
 ; to other programs. The Linker will
 ; resolve external references.
GLOBAL SYM1,SYM2 ; Multiple declarations on the same
 ;line are legal when separated by a
 ;comma. The spaces are ignored.

GLOBALS ON

Defines all labels after **GLOBALS ON** as global labels that other programs can reference. It does not affect Local Labels.

GLOBALS ON

SYM1	NOP	;Declares the labels SYM1 and SYM2,
SYM2	NOP	;accessible to other programs. This
		;directive is not reset by the module
		;and endmod directives. The default
		;is GLOBALS OFF .

GLOBALS OFF

Puts assembly back into its default mode, in which you have to specify a global label with **GLOBAL**.

LABEL:	XREF
	EXTERN
	EXTERNAL

States that a label is defined in another program. Designate multiple labels by separating each with a comma.

LABEL: ASK PROMPT

Outputs 'PROMPT' to the terminal and waits for a 1 character response, subtracts 30 hex from it, and sets LABEL equal to the result. The purpose of this is usually to introduce a 0/1 flag into a program. A carriage return terminates 'PROMPT'. On pass 2, the outputs PROMPT and the response.

**DISK_SIZE: ASK ASSEMBLE FOR 8" (=1) OR 5 1/4" (=0)
DRIVES ? :**

**MESSAGE
MESSG**

Outputs a user-defined message to the screen on pass 2 only.

Assembly Mode Directives

LABEL: SECTION OPTIONS

Lets you generate your own section names. Three sections are predefined: **CODE**, **DATA**, and **PAGE0**. The default is **CODE**. In one file you can have up to 256 section names, each up to 32 characters long. You can nest sections. Names are case-sensitive. An optional decimal point may precede a section name. You can give your own sections a **PAGE0** attribute. After defining a section, you can use the name as a mnemonic to switch the program to and from it. Here are examples of defining section names and switching between sections.

```

NOP                ;This instruction goes into CODE by default
.DATA             ;Switch to predefined DATA section
.BYTE            ;This byte goes into DATA section
SECTION1: .SECTION ;Define a new section. The definition
                ; makes the section active automatically
NOP                ;This instruction goes into SECTION1 section
.CODE            ;Switch back to the section named CODE
NOP                ;This instruction goes into CODE section
.SECTION1        ;Switch to user-defined section SECTION1
NOP                ;This instruction goes into SECTION1
.BYTE            ;Any section may contain code, data or both.

.PAGE0           ;Switch to predefined Page0 section.
VAR1: .DB         ;Define a Page0 variable.
.CODE            ;Switch back to section named Code.
LDAA    VAR1     ;References the Page0 variable.
                ;Direct addressing mode will be used

MY_PAGE0: .SECTION PAGE0 ;Define a new section with
                ; Page0 attribute. Definition makes this
                ;section active automatically.

VAR2:.DB         ;Define another Page0 variable.

.CODE            ;Switch back to section named Code.

LDAA    VAR2     ;This instruction references the second
                ;Page0 variable. Direct addressing
                ;mode will be used.
```

Note: The Linker Operating Instructions describe how the Linker handles section names.

The **OPTIONS** field (optional) lets you tailor linker control. The linker will not prompt for a load offset for any section with specified **OPTIONS**. So you can specify some or all of a link in your source file, using the following keywords:

OFFSET	NNNN	Relocate section to address NNNN
INDIRECT	NNNN	Link indirectly at address NNNN
STACKED		Stack section
REF_ONLY		Use section for reference only

As long as you separate multiple options by a comma, you can specify them in any order. Options are not case-sensitive.

For a description of **Indirect**, **Stacked** and **Reference Only** linking, see Linker Operating Instructions. Here are examples showing how to specify linker options.

SECTION1:	.SECTION	OFFSET 100H
SECTION2:	.SECTION	OFFSET 100H, REF_ONLY
SECTION3:	.SECTION	STACKED
SECTION4:	.SECTION	STACKED, REF_ONLY
SECTION5:	.SECTION	REF_ONLY, STACKED
SECTION6:	.SECTION	INDIRECT 2000H

ENDS

Used with the **SECTION** directive, **ENDS** lets you terminate nested sections in a file.

ABSOLUTE

We have kept this directive mainly for compatibility with Version 3.0 of the assembler. In Version 4.0, the **PAGE0** directive supersedes **ABSOLUTE**, which is now useful only for defining templates. You should always assemble executable instructions in **Relative** mode, which is the assembler's default (see **Absolute versus Relative**).

RELATIVE

Tells the assembler to return from **Absolute** to **Relative** mode. Always assemble executable instructions in **Relative** mode, which is the default.

RADIX VALUE

Specifies the number base for all numerical operations within a program. **VALUE** may be one of the following:

2 or B	=	Binary
8 or O or Q	=	Octal
10 or D	=	Decimal
16 or H	=	Hexadecimal

If you do not specify **VALUE**, **RADIX** defaults to base 10. The assembler then assumes that all other numbers will be flagged with B, Q, or H after the constant. After you specify base 16, you cannot use B or D to define a decimal or binary number, since in hex both B and D are recognized as digits.

LEADZERO ON/OFF

LEADZERO ON, executed at the start of a program, tells the assembler to recognize alphabetic characters in the operand field as hex numerals if they are immediately preceded by a zero. Otherwise the assembler treats these characters as a label name. The default is **OFF**.

LABEL	equ adb	;treated as a label name
LABEL	equ 0adb	;treated as a hex number in ;LEADZERO mode

INCLUDE FILENAME

Includes the named file in the assembly. Filenames may include pathnames. You must completely specify filename extensions. You may not include nested files.

SPACES ON/OFF

SPACES ON tells the assembler to allow instructions containing spaces between operands. As a result, you **MUST** begin comments with a semi-colon, to flag the end of an instruction and the start of a comment. The default is **SPACES OFF**, which disables spaces between operands, so that you need not start comments with a semi-colon.

TWOCHAR ON/OFF

Toggles between enabling and disabling the use of the ascii two character abbreviations shown below. The default is OFF.

"CR" or 'CR'	Carriage return
"LF" or 'LF'	Line feed
"SP" or 'SP'	Space
"HT" or 'HT'	Horizontal tab
"NL" or 'NL'	Null

COMMENT CHARACTER

Lets you write blocks of comments as follows:

COMMENT X ;where X can be any character.

The assembler will treat everything from the first X to the second as a comment block. The assembler does not scan for the second X until the line following the first. Therefore the comment field must be at least two lines long.

ENDMOD

Use this directive with **MODULE** to specify the end of a module in a file.

MODULE

Use this with **ENDMOD** and the **Library Manager**. Normally, libraries consist of many small routines. When the Linker cannot find a **Global** symbol in a file to be linked, it can search the libraries for the symbol it needs, so that each routine must be in a separate file and each file must be assembled separately. Using **MODULE** and **ENDMOD**, however, you can bracket each routine and have them all in one file. Since the assembler then treats each module as a totally separate assembly, you must declare **External** all references to external symbols, and **Global** all symbols used by other modules. A module must end with **ENDMOD**. You cannot nest modules. You can have **include** files inside a module, but not vice versa. A link file can contain as many modules as you like. The assembler produces an output file with the extension **pak**. This you can process most easily with the Librarian's **ADD ALL** and **REPLACE ALL** commands. (See **Librarian Commands**).

```
.MODULE JUMP_TABLE ;Define library name
.GLOBAL JUMP_TABLE ;Make table available to
                    ;other modules in file
.EXTERN ROUTINE1   ;Define externals
.EXTERN ROUTINE2
```

```
JUMP_TABLE: .WORD ROUTINE1 ;Store Routine Addresses
            .WORD ROUTINE2
            .ENDMOD          ;Define end of module
            .MODULE ROUTINE1 ;Define library name
            .GLOBAL ROUTINE1 ;Make routine available
```

```
ROUTINE1:   NOP
            .ENDMOD          ;Define end of module
            .MODULE ROUTINE2 ;Define library name
            .GLOBAL ROUTINE2 ;Make routine available
```

```
ROUTINE2:   NOP
            .ENDMOD          ;Define end of module
            .END              ;Define end of file
```

If this example file is named **test.asm**, the output file after assembly will be **test.pak**. During assembly you will notice that the assembler **restarts** at the beginning of each module.

Conditional Assembly Directives

IFZ VALUE

If **VALUE** equals zero, **IFZ** assembles subsequent statements until it reaches **ELSE** or **ENDIF**. You can nest conditionals to 248 levels. **VALUE** can be an arithmetic expression, another symbol, or a string.

IF VALUE
IFNZ
COND

If **VALUE** does not equal zero, **IFNZ** assembles subsequent statements until it reaches **ELSE** or **ENDIF**. You can nest conditionals to 248 levels. **VALUE** can be an arithmetic expression, another symbol, or a string.

IFTRUE VALUE
IFNFALSE

IFTRUE is similar to **IFNZ**, but more logical with assembly time comparisons. If the condition is true, **IFTRUE** assembles subsequent statements until it reaches **ELSE** or **ENDIF**. Otherwise it ignores everything before the next **ELSE** or **ENDIF**.

IFNTRUE **VALUE**
IFFALSE **VALUE**

IFNTRUE is similar to **IFZ** and complements **IFTRUE**. If the condition is false, **IFNTRUE** assembles subsequent statements until it reaches **ELSE** or **ENDIF**. Otherwise it ignores everything before the next **ELSE** or **ENDIF**.

IFDEF **LABEL**

Searches the symbol table. If it finds **LABEL**, it assembles subsequent statements until it reaches **ELSE** or **ENDIF**. Otherwise it ignores everything before the next **ELSE** or **ENDIF**.

IFNDEF **LABEL**

Searches the symbol table. If it does not find **LABEL**, it assembles subsequent statements until it reaches **ELSE** or **ENDIF**. Otherwise it ignores everything before the next **ELSE** or **ENDIF**.

IFSAME STRING1,STRING2
IFNDIFF

Compares **STRING1** to **STRING2**, and assembles subsequent statements depending on the result. If the strings are identical, **IFSAME** assembles subsequent statements until it reaches **ELSE** or **ENDIF**. Otherwise it ignores everything before the next **ELSE** or **ENDIF**. Strings come in two kinds: with spaces and without spaces. You must enclose a string with spaces in apostrophes, and enter embedded apostrophes as double apostrophes. You don't need apostrophes for non-space strings. You can only compare strings of the same kind, and you must separate them by a comma. **IFSAME** is useful for comparing macro parameter arguments.

IFSAME 'test string','test string'

IFSAME '2500 A.D.'s','2500 A.D.'s'

IFSAME X,Y

The first example shows strings with spaces. The second has embedded apostrophes. The third, a macro testing for a register, has non-space strings.

IFNSAME STRING1,STRING2
IFDIFF

Compares **STRING1** to **STRING2**, and if the strings are not identical assembles subsequent statements until it reaches **ELSE** or **ENDIF**. Otherwise it ignores everything before the next **ELSE** or **ENDIF**. The syntax rules for forming strings are the same as for **IFSAME**. See **IFSAME** for examples.

IFEXT LABEL

Searches the symbol table for LABEL. If LABEL is an external label, **IFEXT** assembles subsequent statements until it reaches **ELSE** or **ENDIF**. If LABEL is not an external label, **IFEXT** ignores everything before the next **ELSE** or **ENDIF**. If **IFEXT** cannot find LABEL, it generates an error message.

IFNEXT LABEL

Searches the symbol table for LABEL. If LABEL is not an external label, **IFNEXT** assembles subsequent statements until it reaches **ELSE** or **ENDIF**. If LABEL is an external label, **IFNEXT** ignores everything before the next **ELSE** or **ENDIF**. If **IFNEXT** cannot find LABEL, it generates an error message.

IFABS LABEL
IFNREL

Searches the symbol table for LABEL. If LABEL is absolute (non-relocatable), **IFABS** assembles subsequent statements until it reaches **ELSE** or **ENDIF**. If LABEL is relocatable, **IFABS** ignores everything before the next **ELSE** or **ENDIF**. If **IFABS** cannot find LABEL, it generates an error message. External labels are relocatable.

IFREL LABEL
IFNABS

Searches the symbol table for **LABEL**. If **LABEL** is relocatable, **IFREL** assembles subsequent statements until it reaches **ELSE** or **ENDIF**. If **LABEL** is not relocatable, **IFREL** ignores everything before the next **ELSE** or **ENDIF**. If **IFREL** cannot find **LABEL**, it generates an error message. External labels are relocatable.

IFMA EXP

Used inside a macro, **IFMA** scans the macro call line for the argument number specified by the value of **EXP**. If it finds the argument, it assembles subsequent statements until it reaches **ELSE** or **ENDIF**. If you specify **EXP = 0**, **IFMA** will not find an argument, but so long as there are no arguments in the macro call line, it will assemble subsequent statements. (See **Macros** for examples of **IFMA**.)

IFNMA EXP

IFNMA checks the macro call line for the number specified by the value of **EXP**. If **IFNMA** does not find the argument, it assembles subsequent statements until it reaches **ELSE** or **ENDIF**. If you specify **EXP = 0**, **IFNMA** will detect any argument, and if there is at least one it will assemble subsequent statements.

ELSE

Defines the next statement to be assembled if an **IF** result is false.

ENDC**ENDIF**

Defines the end of a conditional assembly block. Unmatched **IF** - **ENDIF** pairs, will produce an error message. Since a recursive macro is almost always controlled by an **IF** directive, it may require **IFCLEAR**. For while **ENDIF** always executes, **IFCLEAR** does not execute when an **IF** result is false.

IFCLEAR

IFCLEAR does not execute when an **IF** result is false. In a recursive macro, **IFCLEAR** keeps **IF** - **ENDIF** pairs balanced (so that the macro can eventually terminate), but it keeps program control under full control of **IF** checking. So, since some kind of **IF** almost always controls a macro exit, **IFCLEAR** is a good way to enforce full condition-checking when a macro contains **MACEXIT** for early exit.

EXIT**MESSAGE**

If executed inside a conditional, **EXIT** terminates assembly. The assembler then outputs **MESSAGE** as a user-defined error message of up to 79 characters. If the surrounding condition is true, **EXIT** will execute. If the surrounding condition is false, **EXIT** does not execute and assembly continues uninterrupted. Since **EXIT** terminates assembly on the first pass, the assembler will not create a listing file.

```
IFTRUE  TABLE_SIZE .UGT MAX_TABLE_SIZE  
EXIT    "MAXIMUM TABLE SIZE REACHED"  
ENDIF
```

Listing Control Directives

LIST ON

LIST

If you set **LIST ON/OFF** (option **L**) as listing destination when you enter the assembler, use **LIST ON** to activate listing. Since the assembler defaults to **LIST OFF**, use **LIST ON** if you want to list only flagged sections.

LIST OFF

NOLIST

NLIST

If you set **LIST ON/OFF** (option **L**) as listing destination when you enter the assembler and include **LIST ON** in your source program, you can cancel listing with **LIST OFF**. This is the default so you need only use it after **LIST ON**.

MACLIST ON

MLIST

Turns on listing of macro expansions. This is the default.

MACLIST OFF

MNLIST

Turns off listing of macro expansions. The default is **ON**.

CONDLIST ON

Turns on listing of false conditional assembly blocks. This is the default.

CONDLIST OFF

Turns off listing of false conditional assembly blocks. The default is **ON**.

ASCLIST ON

Turns on listing of ascii strings that require more than 1 line of listed object code. This is the default.

ASCLIST OFF

Turns off listing of ascii strings that require more than 1 line of listed object code. The assembler will list only the first line of object code. The default is ON.

PW EXP

Sets the printer page width. The default is 132 columns.

PL EXP

Sets the printer page length. The assembler issues a form feed when the defined limit is reached. If it finds an error, the assembler outputs the error message before the form feed. The default page length is 61 lines.

TOP EXP

Sets the number of lines from the top of the page to the page number. The default is zero.

PASS1 ON

Turns on listing of pass 1. This is useful for finding the kind of errors that occur when the assembler takes different paths on passes 1 and 2. (The usual error message for this problem is **Symbol value changed between passes**). **PASS1 ON** is useful also for finding nested conditional assembly errors.

PASS1 OFF

Turns **OFF** Pass 1 listing if it has been **ON**. This is the default.

PAG**PAGE****EJECT**

Outputs a form feed to the listing device.

NAM
STRING
TTL
TITLE
HEADING

Prints **STRING** at the top of every page. If you don't specify **STRING**, **TITLE** will not execute. You can change the title as often as you like and can turn it off at any time. Maximum title length is 80 characters. If there are tabs between **TITLE** and the start of the string, the first two will be ignored. All subsequent spaces and tabs will be included in the title. **TITLE** does not take effect till the page after you execute it. To force it to print before the code that follows it, use **PAGE** immediately after **TITLE**.

STTL **STRING**
SUBTTL
SUBTITLE

Prints **STRING** at the top of every page. If you define a title, the subtitle will appear below. If there is no title, the subtitle will be there alone. If you don't specify **STRING**, **SUBTITLE** will not execute. You can change the subtitle as often as you like and can turn it off at any time. Maximum subtitle length is 80 characters. As with **TITLE**, the first two tabs between **SUBTITLE** and the start of **STRING** are ignored and subsequent spaces and tabs included in the subtitle. Since **SUBTITLE** also takes effect on the page after you execute it, use **PAGE** immediately after **SUBTITLE** if you want to force it to print immediately.

Linker Control Directives

FILLCHAR VALUE

SECTION and **ORIGIN**, since they reset the program counter, can leave gaps between program memory areas. **FILLCHAR** tells the linker, when linking executable output only, to fill these gaps with the **VALUE** you specify. All other output formats begin a new record when there is such a gap.

RECSIZE VALUE

Lets you change the default lengths for Intel Hex and Motorola S records. **VALUE** replaces Intel Hex's 32 data byte and Motorola S's 16 data byte defaults.

SYMBOLS

SYMBOLS with the **m**, **n**, or **z** link-time options, is the only way to tell the linker to send MicroTek, Extended MicroTek and Zax symbol-table formatted output to the output file.

OPTIONS OPTION LIST

Selects the options for linking. You can change the default output filetype (Intel Hex) by using the **OPTION LIST**. For the list of options see **Linker Options**.

LINKLIST

Tells the linker to relocate the assembler listings so that the execution address, object-code field addresses and cross-reference table values are the actual run-time values. **LINKLIST** works only when you select **Listing to Disk**.

COMREC STRING

Lets you insert a comment record in Motorola outputs.

Assembly Time Operations

Calculations

Here is a list of permitted assembly-time calculations and their priority levels. Priority 7 operations are performed first. Use parentheses to force a different order. All calculations except exponentiation use 80 bit integer arithmetic. Exponentiation uses an 8 bit exponent. The maximum number of pending operations is 16.

Operation	Priority	Description
Unary +	7	Optionally specifies a positive operand.
Unary -	7	Negates the following expression.
\ or .NOT.	7	Complements the following expression.
Unary >	7	Keeps the high order byte of the following address. This must be used to obtain relocatable byte address values.
Unary <	7	Keeps the low order byte of the following address. This must be used to obtain relocatable byte address values.
**	6	Unsigned exponentiation
*	5	Unsigned multiplication
/	5	Unsigned division
.MOD.	5	Remainder
.SHR.	5	Shift the preceding expression right (with 0 fill) the number of times specified in the following expression.
.SHL.	5	Shift the preceding expression left (with 0 fill) the number of times specified in the following expression.
+	4	Addition
-	4	Subtraction
& or .AND.	3	Logical AND
^ or .OR.	2	Logical OR
.XOR.	2	Logical exclusive OR

Comparisons

Assembly time comparisons all return **1** if the comparison is true and **0** if it is false. The comparisons are:

=	or	.EQ.	-	Equal
>	or	.GT.	-	Greater than
<	or	.LT.	-	Less than
		.UGT.	-	Unsigned greater than
		.ULT.	-	Unsigned less than

PAGE 0

PAGE0 predefined section lets you define values in the range 00h to FFh. In a PAGE0 section, if you define an external value it will have a PAGE0 value. In PAGE0 you can define values as external by using PAGE0 as a modifier in front of every label declared external. When relocating values declared in a PAGE0 section, the linker checks them so as not to relocate them out of the page. Here are some examples:

```

        .PAGE0
VALUE:  DB      12h
        .EXTERNAL LABEL ;give LABEL a PAGE0 attribute
        .EXTERNAL PAGE0 LABEL
                                ;give LABEL a PAGE0 attribute

        .EXTERNAL PAGE0 LABEL1,PAGE0 LABEL2
                                ;give LABEL1 and LABEL2
                                ;PAGE0 attributes

SECTION1:SECTION PAGE0
                                ;define SECTION1 and give it
                                ; a PAGE0 attribute
        DB      23h           ;the byte is a PAGE0 value
        CODE                                ;CODE is now the current section
        NOP                                ;this is in the CODE section
        SECTION1                    ;SECTION1, the PAGE0 section,
                                ;is now the current section
LABEL:  RMB                                ;this label, defined in SECTION1,
                                ;can't be relocated out of PAGE0

        END

```


Absolute Versus Relative

You can use the **ABSOLUTE** directive to give a symbol an absolute value. We support this directive to maintain compatibility with our series 3.0 assemblers. If you use **ABSOLUTE**, you must use **RELATIVE** to return the assembler to relocatable mode. The assembler should always be in Relative mode to assemble executable instructions. Absolute & Relative attributes do not change when the section is changed.

ABSOLUTE is also useful for laying out assembly language structures. You can do this in a user-defined section, as follows:

```

STRUCTURE_SECTION:      .SECTION
                           .ABSOLUTE
                           .ORIGIN    <initial offset>
NAME:                   .DS        <expression>
COMPANY:                .DS        <expression>
ADDRESS:                .DS        <expression>
CITY:                   .DS        <expression>
STATE:                  .DS        <expression>
ZIP_CODE:               .DS        <expression>
STRUCTURE_SIZE: .DS              0

```

If the "initial offset" for the structure is zero and "expression" equals the size of the corresponding member of the structure, you can leave out the **ORIGIN** statement. The next example reserves storage space for three different structures of this type:

```

STRA:                   .DS          STRUCTURE_SIZE
STRB:                   .DS          STRUCTURE_SIZE
STRC:                   .DS          STRUCUTRE_SIZE

```

Forming structures in this way has the advantage of automatically computing offsets and structure sizes while letting you add or delete structure elements without having to re-compute the offset for each member. If you link this section as a "reference only section" by preceding the "load offset" given at link time with a hyphen, then the bytes reserved by the **DS** directives will not be included in the output file. For more information on "reference only", see **2500 A.D.Linker**.

Note. The correct procedure for generating executable code at absolute addresses is:

- 1) assemble in relative mode
- 2) give the linker a load offset of zero for these instructions at link time.

If executable instructions are assembled in Absolute mode, relative references will be calculated with absolute values. Displacements will therefore be an absolute number, just as if the symbol was defined with the **EQUAL** directive.

Consider the following example:

```
                .CODE  
                .ABSOLUTE  
                .ORG      20H  
LABEL:  NOP  
        BRA      LABEL  
        .END
```

where LABEL has a value of 20H. The BRA instruction will use +20H as its displacement value. This is equivalent to the instruction:

```
BRA      20H
```

where the next instruction executed will always be +20H bytes away from the BRA instruction.

Macros

Definition

A macro is a sequence of source lines that will be substituted for a single source line. You must define a macro before you can use it. The assembler stores the definition and, when it reaches the macro name, substitutes the defined source lines. A macro definition may include arguments, substituted into any field except the comment field. Dummy arguments may not contain spaces.

For actual macro calls, arguments may be of any type; direct, indirect, character string or register. No argument except an Ascii string bracketed by apostrophes may include spaces. (Apostrophes in the string must appear as double apostrophes.) So long as the dummy argument names are identical, arguments can be passed through to nested macros. Memory space is the only limit on macro nesting.

A macro must be defined by **.MACRO** and must end with **.MACEND** or **.ENDM**. The macro's name belongs in the label field.

Argument Separators

In the macro call line, arguments must be separated by commas. Leading spaces and tabs are ignored. If there is no argument, a single comma acts as a place holder.

The asterisk (*) as a macro argument is a multiplication sign, not the program counter. In a macro body, valid argument separators are:

```
, + - * / ** \ & ^ = ( ) [ ] |
.NOT. .AND. .OR. .XOR. .EQ. .UGT. .LT .UGT.
.ULT. .SHR. .SHL.
```

Labels

Macro definitions can contain labels, which may be defined as explicit or implicit. Explicit labels in the macro definition will not be altered by the Assembler. Implicit labels are followed by a #, for which the assembler substitutes a 4-digit macro expansion number. An implicit label and its macro expansion number must not exceed 32 characters. An argument can specify a label.

String Concatenation

The broken bar character (| = hex 7C) is the string concatenation operator. You can concatenate only inside a macro.

Value Concatenation

You can concatenate a string and an expression value with the broken bar character (`|` = hex 7C) followed by a left angle-bracket, the expression, and a right angle-bracket. There must be no spaces between the broken bar and the left angle- bracket.

CONCAT	.MACRO	ARG
VALUE:	.VAR	VALUE+1
ARG <VALUE*2>	.EQU	31
	.ENDM	
VALUE	.VAR	0
CONCAT	LABEL	

The invocation **CONCAT LABEL** will produce:

LABEL2	.EQU	31
---------------	-------------	-----------

It is important to initialize **VALUE** before invoking the macro. Otherwise the **LABEL** will have different values on passes 1 and 2.

Mnemonic Definitions

The assembler searches tables in the order:

- 1st - Mnemonic Table**
- 2nd - Macro Definition Table**
- 3rd - Assembler Directive Table**
- 4th - Section Name Table**

To redefine a mnemonic use **MACFIRST**. This switches the search order, putting Macro Definition Table first and Mnemonic Table second.

Macro Examples

The first example does string comparisons.

```

CMP_STRING: .MACRO ARG1
    IFNMA 1
    EXIT "CMP_STRING needs an argument"
    MACEXIT
    ENDIF
    IFSAME "JANUARY",ARG1
MONTH BYTE 1
    MACEXIT
    ENDIF
    IFSAME "FEBRUARY",ARG1
MONTH BYTE 2
    MACEXIT
    ENDIF
    IFSAME "MARCH",ARG1
MONTH BYTE 3
    MACEXIT
    ENDIF
    IFSAME "APRIL",ARG1
MONTH BYTE 4
    MACEXIT
    ENDIF
    IFSAME "MAY",ARG1
MONTH BYTE 5
    MACEXIT
    ENDIF
    IFSAME "JUNE",ARG1
MONTH BYTE 6
    MACEXIT
    ENDIF
    EXIT "argumnet error in macro string"
    ENDM
CMP_STRING "APRIL"
END

```

The next demonstrates argument substitution in a macro operand field.

```
EMPLOYEE_INFO: .MACRO      ARG1,ARG2,ARG3
NAME:          .DB         ARG1
DEPARTMENT:    .ASCII      ARG2
DATE_HIRED:    .LONG       ARG3
               .ENDM
EMPLOYEE_INFO 'JOHN DOE',PERSONNEL,101085
               .END
```

Here is the previous example, changed so as to pass the argument into the label field. This lets you alter the program structure.

```
EMPLOYEE_INFO: .MACRO      ARG1,ARG2,ARG3
ARG1:          .DS         30H
ARG2:          .DS         10H
ARG3:          .LONG
               .ENDM
EMPLOYEE_INFO NAME,DEPARTMENT,DATE_HIRED
               .END
```

You can also substitute into the mnemonic field, and use the # sign to generate a label within the macro.

```
INSTRUCTION:   .MACRO ARG,VAL
ARG
LAB#:          .DS        VAL
               .MACEND
INSTRUCTION    NOP,7
               .END
```


MACFIRST ON/OFF

To redefine a mnemonic **MACFIRST ON** must precede the new macro definition.

```

NOP:      MACFIRST    ON
          .MACRO  ARG
          .DB      ARG
          .ENDM
          NOP      FFH
          .END
    
```

MACDELIM

MACDELIM passes commas into a macro. The following examples show the syntax:

```

DELIM_EX: MACDELIM    {
          MACRO      ARG1,ARG2
          BYTE      FFH,ARG1 ARG2
          ENDM
          DELIM_EX   {A4H},{,12H}
    
```

```

DELIM_EX: MACDELIM    [
          MACRO      ARG1
          BYTE      FFH ARG1
          ENDM
          DELIM_EX   [,A4H]
    
```

```

DELIM_EX: MACDELIM    (
          MACRO      ARG1
          BYTE      FFH ARG1
          ENDM
          DELIM_EX   (,A4H)
    
```

*Recursion***MACEXIT**

Here is a recursive macro that reserves a number of data bytes defined by dummy argument ARG1 and fills them with the value specified by the arguments ARG2,ARG3,ARG4,ARG5,ARG6. The macro also shows the use of **MACEXIT** and **IFCLEAR**. Each successful loop execution decrements the count. The statement **RESERVE** and its subsequent arguments recall the macro.

```

RESERVE: .MACRO      ARG1,ARG2,ARG3,ARG4,ARG5
COUNT:  .VAR        ARG1
          .IFZ        COUNT
          .IFCLEAR
          .MACEXIT
          .ENDIF
COUNT:  .VAR        COUNT-1
          .BYTE        ARG2,ARG3,ARG4,ARG5
          .RESERVE     COUNT,ARG2,ARG3,ARG4,ARG5
          .MACEND

```

This macro is called initially by a statement such as:

```

RESERVE 10,AH,BH,CH,DH,EH      ;Fill 50 bytes with the
                                ;sequence ABCDE.

```

Notes: It is perfectly legal for a recursive macro, like the one in the last example, to call another recursive macro and so on to any level you like. Also, note the use of **IFCLEAR** to maintain the conditional **IF - ENDIF** pair balance. **IFCLEAR** is not strictly necessary here, since **MACEXIT** returns all conditionals to their original state.

Assembler Error Messages

A Label Is Illegal On This Instruction

Flags labels that can not receive a relocation value, such as ENDM or MACEND. Thus, the label is not allowed for the instruction.

Attempted division by Zero

Divisor operand evaluated to 0.

Can't Create Output File - Disk May be Full

The disk may actually be full, or the operating system may be letting too few files be open at one time. See System Requirements to correct this error.

Can't Open Input File

The operating system is letting too few files be open at one time. See System Requirements to correct this error.

Can't Find Filename .OBJ

The .OBJ filename does not exist or the operating system is letting too few files be open at one time. See System Requirements to correct this error.

Can't Recognize Number Base

The number base specified is not one the assembler accepts.

Can't resolve Operand

Can't tell what the programmer intended.

'Endmod' Can't Be In 'Include' File

Extra Characters at End of Operand

Usually a syntax or format error. This error is the last check on any instruction before the Assembler proceeds to the next line. It indicates that there are extra characters after a legal operand terminator.

Hex # And Symbol Are Identical

There is a label identical to a hex number that is being used as an operand. (Even the hex number indicator must be in the same place for this error to be generated.)

Illegal Addressing Mode

Can't address the operand using this form.

Illegal Argument

You can't use this operand here.

Illegal Ascii Designator

Bad punctuation on Ascii character.

Illegal External References

External reference can't be used here.

Illegal Label 1st Character

Labels must start with an alpha character.

Illegal Local Labels

You are using labels that can't be defined as local. For example
.VAR.

Illegal Mnemonic

Mnemonic doesn't exist and wasn't defined as a Macro.

Illegal Nested Include

One included file contains an .INCLUDE directive. This error may also indicate that an included file did not have an END statement.

Illegal Register

The specified register is not legal for the instruction.

Label Value Changed Between Passes

Symbol value decode during pass 1 not = pass 2. This error is usually caused by the Assembler taking different paths on Pass 1 as compared to Pass 2 due to conditional directive arguments changing value. The directive PASS1 ON/OFF can be useful in finding these types of errors.

Macro Name Must Appear On Same Line As Macro Definition**Macro Stack Overflow**

Macros are nested too deeply. This error can be caused by too many recursive macro calls. The stack has room for approximately 700 nested or recursive macro calls. The number of calls is affected by the number of arguments the macro uses.

Maximum External Symbol Count Exceeded

There were too many externals in a module. The maximum is about 500 externals per module.

Missing Delimiters on Macro Call Line

Unmatched delimiters when a macro was invoked.

Missing Endmod Directive**Multiple Externals in the Same Operand**

More than one external exists in the same operand.

Missing Label

A label is required for this instruction.

Missing Module Directive**Missing Right Angle Bracket**

Right angle bracket is mandatory.

'Module' Can't Be In 'Include' File**Multiply Defined Symbol**

Symbol defined previously (not including '.VAR').

Must Be In Same Section

The instructions operand is in a different section.

Nested Conditional Assembly Unbalance Detected

Any '.IF' type instruction without a matching '.ENDIF'.

Nested Section Unbalance

A nested section definition without an ENDS.

Not Enough Parameters

More arguments than parameters in a macro. See ARGCHK directive

Non-Existent Include File

The include file could not be found.

Operand Must Be Defined As An 8 Bit Relocatable Value

This occurs when a 16 bit address is used in an 8 bit instruction.

The

< or > sign must be used to make the value relocatable.

Relative Jump Too Large

Destination address is in a different page.

Syntax Error

Usually a missing comma or parenthesis.

Undefined Symbol

Symbol wasn't defined during pass 1.

Too Large

The destination is too small for the operand.

LINKER

LINKER

2500 A.D. Linker - Version 4.05

Introduction

With the 2500 A.D. Linker you can write modular assembly language programs. The Linker can resolve external references and relocate addresses. It generates all the most used file formats, so that you do not need a separate format-conversion utility.

Except when generating executable output, the Linker runs entirely in RAM. You can link a file of any size, so long as there is enough memory. For an executable file, the Linker creates as many scratchpad files as it needs to sort program sections into ascending order.

Each object file may have up to 256 user-defined sections. The Linker can search up to 50 library files to resolve external symbol references. It can process a combination of 256 input files and library modules, and 256 different section names. There is no size limit on each section.

You can specify a linking address in a file, or relocate it at link time. You can set link controls for sections as you define them in the source file. For example, you can define a section as reference only, so that, though the linker includes its link information, it does not appear in the output file (see **SECTION** and **OPTION** directives). Or you could link a section at different run-time and load addresses, generating ROM-able code that moves to read/write memory at run time (see **Indirect Linking**).

When you specify listing to disk, **LINKLIST** relocates the listing at the actual run-time addresses (see **Linker Control**).

You can invoke the Linker in Prompt, Command Line, or Data File mode; select output format by source-file directive or with one of the Linker options; and save on disk the Load Map, an alphabetic global symbol list, and all link errors.

The Linker can output several types of symbol-table files: relocated 10 character global symbols; relocated 32 character global symbols; and Zax and Microtek formats, both of which include all symbols.

Operating Instructions

Prompt Mode

Input

To run the Linker in prompt mode, type **link**. You will see the prompt **Input Filename :**. If you don't include a filename extension, the Linker's default is **.obj**.

The Linker opens the input file, then prompts **Enter Offset for 'section name'**. Predefined sections come first, followed by each program section that has a non-zero size, in the order in which you defined them. The Linker adds the offset to the value of any **ORG** statements in the source file to arrive at the load address. (See **Linker Examples** for input illustrations.)

If you put a minus sign **before** an offset, the Linker treats the section as reference only, relocating it but not including it in the output file. If you put a semi-colon **after** an offset, the Linker relocates the section and automatically stacks every later section immediately on top of the one before it.

If you enter only a carriage-return at a section-offset prompt, the Linker stacks that section directly on top of the one before. If there are more sections in the file, the Linker prompts for the next section offset. If there aren't, it prompts for the next input filename.

If you enter only a carriage-return at an input-filename prompt, you will see the prompt **Output Filename :**.

Output

If you enter only a carriage-return in response to the output filename prompt, the Linker generates an output file with the same name as the first input file. The filename extension depends on the type of output file you select.

After you have entered the output filename, the Linker prompts for library filenames. If you enter only a carriage-return in response to the library-filename prompt, the Linker assumes there are no more library files, and prompts for Linker options. (See **Linker Options**).

Data File Mode

Data File mode is useful with large or complex links. It is much the same as Prompt Mode, but writes all responses to the prompts in a file which it then submits to the Linker. Invoke Data File mode by entering:

Link data_file

The Linker reads the file **data_file.lnk** and uses the responses in the file, line by line. The default data-file extension is **.lnk**. Since carriage-return only responses may be difficult to see in a data file, you can type an underbar ('_') before the carriage-return to improve readability. Linker options go at the end of the file, just as in prompt mode.

Example:

file1	First input filename
2000	Put the CODE section at 2000H
4000	Put the DATA section at 4000H
file2	Second input filename
_	Stack CODE section on top of 1st CODE section
_	Stack DATA section on top of 1st DATA section
_	No more input filenames
_	Use default output filename
_	No library filenames
d3	Create disk file & Motorola S37 file

This Data File will link 2 files with the CODE section starting at 2000h and the DATA section starting at 4000h. The Linker will use the default output filename and the D and 3 options, generating a disk map file and a Motorola S37 output file.

The easiest way to construct a Data File is to run through the link process in Prompt mode, writing down each response. Then, use a text editor to create a file with each response on a line by itself. This file should have the extension **.lnk**.

A line with an asterisk in column 1 is a comment.

Command Line Mode

You can invoke the Linker from the command line. The command line form is as follows (bracketed arguments are not required):

Link [-q] -c file1 [-lnnnn] file2 [-lnnnn] -L [-ofile] [-options]

The **-q** argument puts the Linker in Quiet mode, in which it outputs nothing except link errors to the terminal.

The **-c** argument sets the Linker to Command Line mode.

file1 and **file2** are input files. **-l** precedes an offset address for each file. If you omit the offset, the Linker stacks a file on top of the previous one, placing matching section on top of matching section.

-o precedes an output filename. If you omit the output filename, the Linker will create an output file with the same name as the first input file. The extension depends on the output-file format.

Use the **-L** option to specify library filenames. The Linker accepts up to **50** library filenames.

The **options** field lets you specify Linker options. You must begin the list with a minus sign, after which you can enter all the options you need. (See **Linker Options** for the list of options)

Linker Options

Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <cr> = Default)

- D** Create a disk file containing all link errors, an alphabetized global symbol table and the load map. This file has the same name as the Linker output file, with the extension **.map**.
- S** Create a symbol file to use in debugging. The file contains all global symbols and relocated values. It stores 32 significant characters for each symbol.
- A** Create a symbol file to use in debugging, but limit symbols to their first 10 characters. This option exists to maintain compatibility with the 3.0 version of the Linker.
- M** Create a symbol file in Microtek Format, to use in debugging. This file includes all symbols, local and global. To generate this format, you must include the **SYMBOLS** directive in your source file.
- N** Create a symbol file in Extended Microtek Format, to use in debugging. This format is the same as Microtek Format, with one extra byte for each symbol. It is used to create symbol files for the 2500 A.D.Simulator.
- Z** Create a symbol file to use in debugging in Zax format. This file includes all symbols, local and global. To generate this format, you must include the **SYMBOLS** directive in your source file.
(For all tables, see **Linker Output - Symbol Tables**)

- X** Generate an Executable output file.
- H** Generate an Intel Hex output file.
- E** Generate an Extended Intel Hex output file.
- T** Generate a Tektronix Hex output file.
- 1** Generate a Motorola S19 output file.
- 2** Generate a Motorola S28 output file.
- 3** Generate a Motorola S37 output file.
- <cr>** The default output format

At any one time, only three options can be active:-

D

one of **S, A, M, N, or Z**

one of **X, H, E, T, 1, 2, or 3.**

If you enter more than one symbol file or output file option, the Linker accepts only the last entry.

Address Relocation

In relocating addresses, the Linker adds the offset address to whatever address the assembler has generated.

The Assembler keeps a table of attributes associated with each symbol used in the program. If an address-label simply precedes an instruction, then it is relocatable. If the label is defined in an **.EQUAL** directive, then the operand field type determines whether it is relocatable or not. If the operand contains no relocatable tokens, or more than one, then the label is not relocatable. If it contains only one relocatable token, then the label is relocatable.

Byte values cannot be relocated unless you use **>** (unary greater than) for the high byte and/or **<** (unary less than) for the low byte. These operands follow the same relocation rules as full 16 bit address values.

2500 A.D.Linker - Version 4.04

Linker Examples

These examples illustrate how to use the Linker. The examples use **<cr>** (carriage return) only when a prompt requires no other response. All other inputs must end with a carriage return, which the examples omit. Responses to prompts are boldfaced.

In creating a Linker Data File, use exactly the same responses as in Prompt mode.

Single File Assembled At Desired Run Address

A1:

You have assembled one file, setting the run address with the **ORIGIN** directive. You want Executable output, and no Linker options. You have no additional sections, and you do not need to switch between predefined sections. The Linker prompts and responses are:

```
Input Filename : filename  
Enter Offset for 'CODE' : 0  
Input Filename : <cr>
```

```
Output Filename : <cr>
```

```
Library Filename: <cr>
```

```
Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) : x
```

Notes:

The Linker will read the file **filename.obj**, add 0 to all relocatable addresses, and output a file with the name **filename.tsk**.

A2:

This is the same as before, except that you want the output to be in Intel Hex format. The example uses the **h** option. The default format option depends on the processor you are using. For the Z80 it is Intel Hex.

```
Input Filename : filename
                Enter Offset for 'CODE'           : 0
Input Filename : <cr>

Output Filename : <cr>

Library Filename : <cr>

Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) : h
```

Notes:

You can change the default output option with the **OPTIONS** directive.

A3:

This the same as Example 2, except that here you want to create a disk Load Map file, enter your own output filename, and use a library to search for unresolved external references. You can specify the options in any order.

```
Input Filename : filename
                Enter Offset for 'CODE'           : 0
Input Filename : <cr>

Output Filename : user_filename

Library Filename : lib_filename

Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) : hd
```

Single File With Multiple Sections

B1:

You want to link a single file with multiple sections, using the predefined **CODE** and **DATA** sections and stacking the **DATA** section on top of the **CODE** section.

```
Input Filename : filename
      Enter Offset for 'CODE'           : 0
      Enter Offset for 'DATA'           : <cr>
Input Filename : <cr>

Output Filename : <cr>

Library Filename : <cr>

Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) : <cr>
```

B2:

Instead of the predefined sections **CODE** and **DATA**, you want user-defined sections **Program_section1** and **Program_section2**, with **Program_section2** stacked on top of **Program_section1**.

```
Input Filename : filename
      Enter Offset for 'Program_section1' : 0
      Enter Offset for 'Program_section2' : <cr>
Input Filename : <cr>

Output Filename : <cr>

Library Filename: <cr>

Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) : <cr>
```

Notes:

Since the Linker stacks sections in order of definition, there is no way to stack **Program_section1** on top of **Program_section2**. If you need to reverse the order, then you must change the definition order of the **SECTION** directives in the source file.

B3:

This is the same as Example B2, except that you want your sections relocated so that **Program-Section1** will run at 2000h and **Program_section2** at 4000h.

```
Input Filename : filename
      Enter Offset for 'Program_section1'      2000
      Enter Offset for 'Program_section2'      4000
Input Filename : <cr>
```

```
Output Filename : <cr>
```

```
Library Filename : <cr>
```

```
Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) :<cr>
```

Notes:

Addresses are always hexadecimal. With an Executable output file, the Linker fills the gap from the end of **Program_section1** to the start of **Program_section2** with the default fill character FFh. You can change the fill character default with the **FILLCHAR** directive.

Multiple Files With Multiple Sections

C1:

You want to link **file1** and **file2** using **CODE** and **DATA** sections, with **file1** starting at 0.

Input Filename : **file1**

Enter Offset for 'CODE' : **0**

Enter Offset for 'DATA' : **<cr>**

Input Filename : **file2**

Enter Offset for 'CODE' : **<cr>**

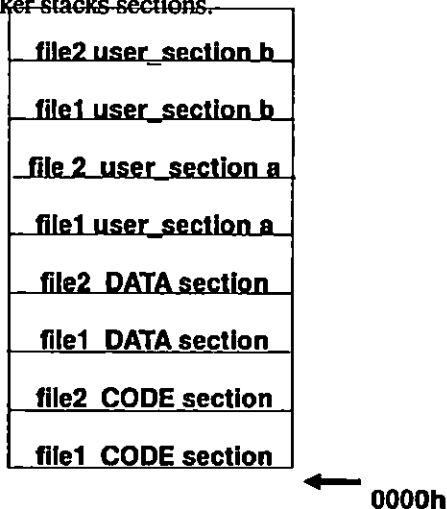
Enter Offset for 'DATA' : **<cr>**

Output Filename : **<cr>**

Library Filename : **<cr>**

Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) : **<cr>**

Note: This will produce the default output file type. The diagram shows how the Linker stacks sections.



Note: You cannot stack **'DATA'** before **'CODE'**. If you need to stack data before code, you have to put your code in a user-defined section instead of the predefined **'CODE'** section.

C2:

You want to stack **CODE** and **DATA** at specific addresses, with **CODE** starting at E000h and **DATA** at 1000h.

Input Filename : **file1**

Enter Offset for 'CODE' : **E000**

Enter Offset for 'DATA' : **1000**

Input Filename : **file2**

Enter Offset for 'CODE' : **<cr>**

Enter Offset for 'DATA' : **<cr>**

Output Filename : **<cr>**

Library Filename: **<cr>**

Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) : **<cr>**

Notes:

Stacking rules are consistent no matter how many input files there are. **You should be particularly careful about using an ORIGIN statement in a section's source-code**, since the Linker will add the ORIGIN value to the last available address and stack the section at the resulting address, regardless of normal section stack-ordering.

Single File With One Reference-Only Section

The Linker relocates a reference-only section so that it can use in the link any globals defined in the section. However, it does not include the section in the output file. Reference-only sections are useful when you have a program resident in ROM or EPROM, data in RAM, and want the output file to contain only the part of the program that is to be stored in ROM.

Define a section as reference-only by putting a minus sign before its address.

In the Load Map, a minus sign before a section name identifies it as reference-only.

The first section in the first file may not be reference-only, since the Linker uses this file as the basis for all other link calculations.

D1:

Your program uses only the predefined **CODE** and **DATA** sections. **CODE** starts at 1000h. You want **DATA** to be stacked on top of **CODE**, used for linking purposes, and then discarded.

```
Input Filename : filename
                Enter Offset for 'CODE'      : 1000
                Enter Offset for 'DATA'      : -<cr>
Input Filename : <cr>
```

```
Output Filename : <cr>
```

```
Library Filename: <cr>
```

```
Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) : <cr>
```

D2:

You want to relocate **DATA** at 4000h and to use it for reference only:

Input Filename : **filename**

Enter Offset for 'CODE' : **1000**

Enter Offset for 'DATA' : **-4000**

Input Filename : **<cr>**

Output Filename : **<cr>**

Library Filename : **<cr>**

Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) : **<cr>**

Indirect Linking

Indirect Linking is the linking of a file section to run at an address other than its load address. This is like phase and dephase; but whereas phase and dephase change the address at assembly time, indirect linking changes it at link time. This can be confusing, and it is easy to generate output with addresses so mixed-up that the program cannot run. Despite this, indirect linking is necessary at times.

Case 1: You have a single board controller application and your program resides in ROM. The data consists entirely of lookup tables or constants. Since it will never be written to, there is no reason to move it out of ROM. You can link normally.

Case 2: You have the same application and program as before, but the data has initialized values that will change as the program runs. You need a start-up routine to move it from ROM to RAM, and must link it indirectly. Indirect linking achieves this by stacking the data normally in ROM but linking it to a run-time RAM address. Use the '@' sign before the load address to tell the Linker to calculate an indirect address.

Direct		Indirect	
Load	Run	Load	Run
(ROM)	(ROM)	(ROM)	(ROM)
CODE DATA	CODE DATA	CODE DATA	CODE
(RAM)	(RAM)	(RAM)	(RAM) DATA

Rules:

- 1) Once you tag a section as indirect, the Linker automatically gives an indirect tag to every section of the same name in all files.
- 2) The Linker stacks indirect sections in the same order as it would if they were not indirect.
- 3) Indirect sections cannot be reference-only, since the whole point is to include the section in the load file.
- 4) Do not link the same section directly and indirectly. It links directly if you do.

E1:

You have three files, **file1.obj**, **file2.obj** and **file3.obj**. Each file has three sections, **program**, **const_data** and **init_data**. You want **program** residing at 0, **const_data** stacked on top of **program**, and **init_data** stored in ROM at power up but running at 1000h in RAM.

```
Input Filename : file1
  Enter Offset for 'program'      : 0000
  Enter Offset for 'const_data'   : <cr>
  Enter Offset for 'init_data'    : @1000
Input Filename : file2
  Enter Offset for 'program'      : <cr>
  Enter Offset for 'const_data'   : <cr>
  Enter Offset for 'init_data'    : @<cr>
Input Filename : file3
  Enter Offset for 'program'      : <cr>
  Enter Offset for 'const_data'   : <cr>
  Enter Offset for 'init_data'    : @<cr>
Input Filename : <cr>
```

Output Filename : <cr>

Library Filename : <cr>

Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) :<cr>

The output file will contain all the sections, stacked in the order:

init_data	(file3)
init_data	(file2)
init_data	(file1)
const_data	(file3)
const_data	(file2)
const_data	(file1)
program	(file3)
program	(file2)
program	(file1)

← 0000h

The Load Map will show the actual load addresses, but the global symbol list will show that all global symbols defined in **init_data** are linked starting at 1000h.

E2:

Before the program can run, the `init_data` section must be moved to location 1000h, and before you can move it, you need to know `init_data`'s size. The easiest and most versatile way to find the size of a section is to bracket it with other sections, even if they are empty. (The Linker does not prompt for a load address if a section is empty, but does if it contains at least one equate.) You can use the first file to set the section link order, by putting code into `file1.asm` as follows:

```

program:                .section
program_addr:           .equal      $
const_data:             .section
const_data_addr:        .equal      $
const_data_end:         .section
const_data_end_addr:    .equal      $
init_data:              .section
init_data_addr:         .equal      $
init_data_end:          .section
init_data_end_addr:     .equal      $

```

program

Sections `const_data_end` and `init_data_end` exist solely to provide addresses for calculating the size of `init_data`. `const_data_end` provides the ROM starting address for `init_data`, and `init_data_end` provides its ending address. The size calculation is then straightforward:

Size of Initialized Data = `init_data_end_addr` - `init_data_addr`

ROM Address of Initialized Data = `const_data_end_addr`

You cannot use `init_data` directly because it has been linked at a different address, and all labels associated with it have been relocated with respect to that address.

E3:

Since the labels that define the starting and ending addresses of **init_data** reside in different sections, a run time startup routine has to do the necessary subtraction to calculate **init_data**'s size.

To enable the Linker to calculate the size of **init_data** correctly, you must define **init_data_end** as an indirect section. Do this by entering **@<cr>** for **init_data_end**'s offset:

```
Input Filename : file1
Enter Offset for 'program'      : 0000
Enter Offset for 'const_data'   : <cr>
Enter Offset for 'const_data_end' : <cr>
Enter Offset for 'init_data'    : @1000
Enter Offset for 'init_data_end' : @<cr>

Input Filename : file2
Enter Offset for 'program'      : <cr>
Enter Offset for 'const_data'   : <cr>
Enter Offset for 'const_data_end' : <cr>
Enter Offset for 'init_data'    : @<cr>
Enter Offset for 'init_data_end' : @<cr>

Input Filename : file3
Enter Offset for 'program'      : <cr>
Enter Offset for 'const_data'   : <cr>
Enter Offset for 'const_data_end' : <cr>
Enter Offset for 'init_data'    : @<cr>
Enter Offset for 'init_data_end' : @<cr>

Input Filename : <cr>
```

Output Filename : <cr>

Library Filename: <cr>

Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) :<cr>

A file linked using indirect sections will usually have all sections stacked on top of the previous one. For convenience, therefore, you can set the Linker to auto-stack mode by putting a semi-colon after any load address or before any **<cr>** response. The Linker then stacks all sections automatically, and stops prompting for offsets. The examples show the three possible positions for semi-colons.

E4:

Input Filename : file1
Enter Offset for 'program' : 0000;
Input Filename : file2
Input Filename : file3
Input Filename : <cr>
Output Filename : <cr>

Library Filename: <cr>

Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) :<cr>

E5:

Input Filename : file1
Enter Offset for 'program' : ;<cr>
Input Filename : file2
Input Filename : file3
Input Filename : <cr>

Output Filename : <cr>

Library Filename: <cr>

Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) :<cr>

E6:

Input Filename : file1
Enter Offset for 'program' : @;<cr>
Input Filename : file2
Input Filename : file3
Input Filename : <cr>

Output Filename : <cr>

Library Filename: <cr>

Options (D, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR>= Default) :<cr>

Linker Output

Symbol Tables

Global Symbol Table

Linker Option: **S**
Symbol Table Filename: **same as Linker output file**
Symbol Table Filename Extension: **.sym.**
Symbol Table File First Byte: **l.d. code E0h.**

The following bytes are repeated for each entry. Positions are relative to the start of each entry:

Bytes	0-31	Global Symbol Name - padded with zeros to fill out the 32 character positions. The entries end with FFh in byte 0.
Byte	32	Most significant byte of relocated Global Symbol value.
Byte	33	Least significant byte of relocated Global Symbol value.
Byte	34	File number in which the Global was defined. The Linker uses this to output the filename along with the value. You may delete this byte or use it for other purposes.
Byte	35	Flag byte - symbol type

Abbreviated Global Symbol Table

Option: **A**
 Symbol Table Filename: **same as Linker output file**
 Symbol Table Filename Extension: **.sym**

This is the table format produced by 2500 A.D Linker, series 4.0.

Bytes	0 - 9	Global Symbol Name - padded with zeros to fill out the 10 character positions. Entries end with FFH in byte 0.
Byte	10	Most significant byte of relocated Global Symbol value.
Byte	11	Least significant byte of relocated Global Symbol value
Byte	12	File number in which the Global was defined. The Linker uses this to output the filename along with the value. You may delete this byte or use it for other purposes.
Byte	13	Flag byte - symbol type.

MicroTek Symbol Table

Linker Option:
Symbol Table Filename:
Symbol Table Filename Extension: .sym

M
same as Linker output file

FEh
Size of Module Name
Module Name
Rest of Module Name
...
Size of Symbol Address
...
Size of Symbol
Symbol Name
High Byte of Address
Low Byte of Address
...
Rest of Symbols and Values
...
FEh
Next Module Information (Same as above)
FFh

Start of Module

3 bytes in length

2 = 16 bits
3 = 24 bits
4 = 8086, 80186, 80286
5 = 32 bits

End of Module

End of File

Extended MicroTek Symbol Table

Linker Option: **N**
 Symbol Table Filename: **same as Linker output file**
 Symbol Table Filename Extension: **.sym**

F0h	Start of Module
Size of Module Name	
Module Name	
Rest of Module Name	3 bytes in length
...	
Size of Symbol Address	2 = 16 bits 3 = 24 bits 4 = 8086, 80186, 80286 5 = 32 bits
...	
Size of Symbol	
Symbol Name	
High Byte of Address	
Low Byte of Address	
Symbol Type	1 byte*
...	
Rest of Symbols and Values	
...	
F0h	End of Module
Next Module Information (Same as above)	
FFh	End of File

* The symbol type values are as follows:

0 = no size	5 = 64 bit double
1 = 8 bit char	6 = 8051 type bit
2 = 16 bit word	7 = 8 bit Z8 type register
3 = 32 bit long	8 = 'ds' label
4 = 32 bit float	9 = equals

Zax Symbol Table

Linker Option: **Z**
Symbol Table Filename: **same as Linker output file**
Symbol Table Filename Extension: **.sym**

\$\$ Module Name	Start of Module
<cr> <lf>	
Symbol Name	
Space	
Symbol Value	
<cr> <lf>	
...	
Rest Of Symbols and Values	
...	
\$\$ Module Name	
<cr> <lf>	
Next Module Information (same as above)	
...	

File Formats

Intel Hex Format

Linker Option: **H**

Record Mark Field signifies the start of a record. It consists of an Ascii colon (:).

Record Length Field two Ascii characters indicating the number of data bytes in this record. The characters are the result of converting the number of data bytes in binary to two Ascii characters, high digit first. An EOF record has two Ascii zeros in this field. There can be no more than 255 data bytes in a record. You can change this with RECSIZE directive.

Load Address Field four Ascii characters, the result of converting the binary value of the address in which to begin loading this record. Order:
 High digit of high byte of address.
 Low digit of high byte of address
 High digit of low byte of address.
 Low digit of low byte of address.
In an end of file record, this field consists of either four Ascii zeros, or the program entry address.

Record Type Field identifies the record type, 00 for data records or 01 for an EOF record. It has two Ascii characters, high digit first, then low digit.

Data Field the actual data, converted to two Ascii characters, high digit first. There are no data bytes in the EOF record.

Checksum Field the 8 bit binary sum of the record length field, load address field, record type field and data field. The sum is then negated (2's complement) and converted to two Ascii characters, high digit first.

Extended Intel Hex Format

Linker Option: **E**

Record Mark Field signifies the start of a record. It consists of an Ascii colon (:).

Record Length Field two Ascii characters indicating the number of data bytes in this record. The characters are the result of converting the number of data bytes in binary to two Ascii characters, high digit first. An EOF record has two Ascii zeros in this field. There can be no more than 255 data bytes in a record, but you can change this with RECSIZE directive.

Load Address Field four Ascii characters, the result of converting the binary value of the address in which to begin loading this record. Order:
High digit of high byte of address.
Low digit of high byte of address
High digit of low byte of address.
Low digit of low byte of address.
In an end of file record, this field consists of either four Ascii zeros, or the program entry address.

Record Type Field identifies the record type, 00 for data records, 01 for an EOF record, or 02 for a segment address .
It has two Ascii characters, the record type high digit first, then low digit.

Data Field the actual data, converted to two Ascii characters, high digit first. There are no data bytes in the EOF record.

Checksum Field the 8 bit binary sum of the record length field, load address field, record type field and data field. This sum is then negated (2's complement) and converted to two Ascii characters, high digit first.

Motorola S19 Format

Linker Option: 1

- Record Type Field** signifies the start of a record. It identifies the record type as follows:
Ascii S1 - Data Record
Ascii S9 - EOF Record
- Record Length Field** specifies the record length which includes the Address, Data and Checksum fields. The 8 bit Record Length value is converted to two Ascii characters, high digit first. The smallest object file record size is 128 bytes, so the Record Length field always consists of 128 data bytes, 2 address bytes and 1 checksum byte, resulting in a record length of 131 bytes. You can change this with RECSIZE directive.
- Load Address Field** four Ascii characters, the result of converting the binary value of the address in which to begin loading this record. Order:
High digit of high byte of address
Low digit of high byte of address
High digit of low byte of address
Low digit of low byte of address
In an EOF record, this field has four Ascii zeros.
- Data Field** the actual data, converted to two Ascii characters, high digit first. There are no data bytes in an EOF record.
- Checksum Field** the 8 bit binary sum of the record length field, load address field and data field. This sum is then complemented (1's complement) and converted to two Ascii characters, high digit first.

Motorola S28 Format

Linker Option: 2

- Record Type Field** signifies the start of a record. It identifies the record type as follows:
 Ascii S2 - Data Record
 Ascii S8 - EOF Record
- Record Length Field** specifies the record length which includes the Address, Data and Checksum fields. The 8 bit Record Length value is converted to two Ascii characters, high digit first.
- Load Address Field** six Ascii characters, the result of converting the binary value of the address in which to begin loading this record. Order:
 High digit of high byte of address
 Low digit of high byte of address
 High digit of mid byte of address
 Low digit of mid byte of address
 High digit of low byte of address
 Low digit of low byte of address
In an EOF record, this field has six Ascii zeros.
- Data Field** the actual data, converted to two Ascii characters, high digit first. There are no data bytes in an EOF record.
- Checksum Field** the 8 bit binary sum of the record length field, load address field and data field. This sum is complemented (1's complement) and converted to two Ascii characters, high digit first.

Motorola S37 Format

Linker Option: **3**

- Record Type Field** signifies the start of a record. It identifies the record type as follows:
 Ascii S3 - Data Record
 Ascii S7 - End of File Record
- Record Length Field** specifies the record length which includes the Address, Data and Checksum fields. The 8 bit Record Length value is converted to two Ascii characters, high digit first.
- Load Address Field** eight Ascii characters, the result of converting the binary value of the address in which to begin loading this record. Order:
 High digit of high byte of high word
 Low digit of high byte of high word
 High digit of low byte of high word
 Low digit of low byte of high word
 High digit of high byte of low word
 Low digit of high byte of low word
 High digit of low byte of low word
 Low digit of low byte of low word
 In an EOF record, this field has eight Ascii zeros.
- Data Field** the actual data, converted to two Ascii characters, high digit first. There are no data bytes in an EOF record
- Checksum Field** the 8 bit binary sum of the record length field, load address field and data field. This sum is complemented (1's complement) and converted to two Ascii characters, high digit first.

LIBRARIAN

LIBRARIAN

2500 A.D. Librarian - Version 4.05

General Description

The Librarian lets you create a library of object modules to link to a program generated by the 2500 A.D. assembler. The linker searches the specified libraries and links in only referenced library modules.

Storage and Memory

The Librarian will process any size file for which you have enough disk space. It needs disk space for an existing library, a temporary file to hold the existing library (so as not to damage it while a new one is being created), and any modules being added to the library. A library can have up to **256** separate modules. The Librarian needs enough memory to store the module directory list and all modules' global symbol records, which it has to check for multiple definition.

Commands

Space and tab are the only valid command line separators. Command descriptions show the full command name, abbreviations and operands. Square brackets (**[]**) enclose optional arguments. Required argument descriptions have boldfaced headings. A carriage-return terminates all commands. Descriptions assume this and omit the carriage-return. Commands are not case-sensitive.

System Defaults

The Librarian system defaults are:

.obj	object filename extension
.pak	packed object filename extension
.lib	library filename extension
.tmp	temporary filename extension

Screen

The Librarian screen displays the modules contained in a library. It highlights the current working module. You can change modules by scrolling through the library directory list, typing **k** or **K** to scroll up, **j** or **J** to scroll down. The directory list can display 16 modules at a time. If you add to the list a module not in the current display, the screen adjusts to display the added module, and highlights it as the current module.

Modules

A module may be a single object file or a module in a packed object file (several object files concatenated). You can manipulate a single module or all modules within a packed object file. The operand **all** or **ALL** acts exclusively on a packed object file. It adds all modules within the file to a library, or replaces all modules in a library. The maximum number of modules is **256**.

See **MODULE** and **ENDMOD** Assembler directives for examples of creating a packed object file.

Installation

The 2500AD Librarian file is on your Assembler distribution disks. The librarian filename for Unix or Ultrix systems is **LIB**. For MSDOS systems it is **LIB.EXE**. Your operating system dictates how you install the Librarian:

STEP 1 - All Systems

Copy the file **lib** or **lib.exe** to the directory and disk drive you want.

STEP 2 - MSDOS Systems Only

The librarian requires the ANSI device driver. If your system defaults do not define **ANSI.SYS** as the display driver, your screen will not display the Librarian properly. If your screen is scrambled, or if it displays non-ASCII characters, you need to add the line

DEVICE = ANSI.SYS

to the **CONFIG.SYS** file on your default drive. If you don't have a **CONFIG.SYS** file in your root directory, you must create one. You must then reboot the system to load the **ANSI** driver.

Your MSDOS Manual has all the details.

Operating Instructions

Command Line Mode:

From the system command line, run the Librarian and read an existing library by typing:

lib filename

Where the filename is an existing library.

To run the Librarian and create a new library, type :

lib filename

Where the filename is the new library name.

Prompt Mode:

You can call the Librarian from the system command line without a filename. Simply type:

lib

The Librarian screen will appear with the message **No Library Currently Open**, and the prompt **Enter Command :**

The next section (**Librarian Commands**) describes all the commands.

Note:

The Librarian is designed for interactive use only. You cannot use it with an MSDOS batch file or Unix shell file.

Librarian Commands

ADD [module /ALL]
A

Function: Adds a new library module or replaces an existing one.

Operand: name of the module to add or **ALL/all**

Notes:

1. **ADD** cannot add a module unless a library is open. If no library is open, **ADD** displays an **Illegal Command** error message.
2. You cannot have more than 256 modules.
3. You can omit the argument, and run the Librarian in prompt mode.
4. You cannot use **ALL** except with a packed object file.

Example 1:

Command with a filename:

Enter Command : **add printf**

Enter Name of Object File : **mylib**

The Librarian will search for file **mylib.obj**. If it does not find **mylib.obj**, it will search for file **mylib.pak**.

Example 2:

Command with **ALL**

Enter Command : **add all**

Enter Name of Object File : **mylib.pak**

The Librarian will add all modules contained in file **mylib.pak**.

Example 3:

Command with argument omitted

Enter Command : **add**

Enter Name of Module : **printf**

Enter Name of Object File :

The Librarian prompts for all responses.

DEL [module]
D

Function: removes a module from the library by deleting its directory entry.

Operand: name of the module to delete

Notes:

1. If you enter a module name, **DEL** searches the library directory list for that module and deletes it.
2. If you omit the module name, **DEL** deletes the current working module (highlighted on screen). It
3. If no library is open, if the library contains no modules, or if the named module is not in the directory list, **DEL** will display the error message **Illegal Command**.
4. The next module in the directory list becomes the new working module, unless the deleted module was the last in the list, in which case the the working module is the one currently last on the list.

Example 1:

Command with module name

Enter Command : **del printf**

The Librarian will search for directory entry **printf** and delete it.

Example 2:

Command with no operand

Enter Command : **del**

The Librarian will delete the current working module.

EXIT

Function: exits the Librarian and returns control to the operating system.

Notes:

If a library is open and contains at least one module, **EXIT** saves it before returning you to the operating system. You will see the message **Saving Library : library name**

FIND [module]

Function: locates a module in a library.

Operand: name of the module to find

Notes:

1. When found, the module becomes the current one.
2. If it cannot find the module in the library, **FIND** displays the error message **Module Not Found In Library**.
3. If you omit the module name, the Librarian prompts for it.

HELP [command name]**H**

Function: displays on-screen help.

Operand: the name of the command for which you want help.

Notes:

1. If you include a command name, **HELP** displays specific information about that command
2. If you omit a command name, **HELP** lists the commands and displays information about the **HELP** command.
3. You will see the prompt **Press Any Key To Continue**.
4. You can use help at any point during Librarian operations, since it removes and then restores directory display.

Example 1:

Enter Command : **HELP**

Displays the main help screen describing how to use the **HELP** command. **Press Any Key To Continue** :

Example 2:

Enter Command : **help ADD**

Displays the help screen describing how to use the **ADD** command. **Press Any Key To Continue** :

LIST module/ALL [disk]

Function: Lists a module's global symbols.

Operand: a module name or **ALL**

Operand: **DISK/disk**

Notes:

1. With a module name as the first operand, the Librarian lists all its global symbols by name in sorted order.
2. With **ALL/all** as the first operand, the Librarian lists the global symbols for all modules in the library.
3. The second operand controls listing destination. **disk/DISK** lists to a disk file. Omitting the operand lists on screen.

Example 1:

Enter Command : **LIST mod2**

Lists module **mod2**'s global symbols on screen.

Example 2:

Enter Command : **LIST ALL disk**

Lists globals for all library modules to a disk file.

NEW [filename]

N

Function: Creates a new library or opens an existing one.

Operand: filename of library to open or create

Notes:

1. If a library is already open and contains at least one module, **NEW** will save it before opening the new one.
2. If you omit a filename, the Librarian prompts **Enter Name of New or Existing Library**.
3. You cannot **add**, **del**, or **rep** modules unless a library is open.

Example:

Enter Command : **new mylib**

The Librarian searches for a library named **mylib.lib**. If it finds one, it displays the library directory on screen. If not, it creates a new library with the name **mylib.lib**.

QUIT

Function: terminates the Librarian and returns control to the operating system.

Notes:

If a library is open and contains at least one module the Librarian will prompt **Do You Want to Save the Library (y/n)**. Typing y or Y will save the library. Any other character will abandon it.

STAT**S**

Function: Displays the status of the current library.

Notes:

1. If a library is open the display screen will show the library name and the number of modules, externals and globals in it. **Press Any Key To Continue**. The screen then restores directory display.
2. If no library is open, you will see the screen message **No Library Open**.

REP
R

[module/ALL]

Function: replaces an existing library module with a new version.

Operand: name of the module to replace or **ALL/all**.

Notes:

1. If you enter a module name, the Librarian prompts **Enter Name of Object File**. Enter the name of the file containing the new module to replace the old one. If the Librarian cannot then find the module you entered, it gives the error message **Undefined Module**. If it cannot find the object file, the message is **Cannot Open File : filename.ext**.
2. If you omit the operand, the Librarian replaces the current working module (highlighted on the screen).
3. If you try to replace a module when no library is open, or when the current library contains no modules, you will see the error message **Illegal Command**.
4. If you enter **ALL**, the command replaces all modules in a packed object file. You can only use **ALL** with packed object files. If you try to use it with other files, you will see an **Illegal Command** error message.

Example 1:

Enter Command : **rep printf**

Enter Name of Object File : **mylib**

The Librarian will search first for the object file **mylib.obj**, next for the module **printf**. It then removes the old **printf** from the current library and adds the new version.

Example 2:

Enter Command : **rep**

Enter Name of Object File : **mylib**

The Librarian will replace the current working module with a new version of the same name from the object file **mylib.obj**

Example 3:

Enter Command : **rep all**

Enter Name of Object File : **mylib**

The Librarian replaces every module in the current library with a new module of the same name from packed object file **mylib.pak**.

TOP

Function: Moves the highlight bar to the first module in the current library directory list.

Notes:

The newly highlighted module becomes the working module.

BOT

Function: Moves the highlight bar to the last module in the current library directory list.

Notes:

The newly highlighted module becomes the working module.

Librarian Error Messages

The Librarian outputs the erroneous command along with the error message.

Cannot Create New Library

The temporary library file could not be renamed. (The write-protect bit may be set wrong. On DOS the library file may be marked "read-only") The name of the file is output with the error message.

Cannot Delete Old Library

An old library file could not be deleted after the new library was created or updated. (The write-protect bit may be set wrong. On DOS the library file may be marked "read-only") The name of the file is output with the error message.

Cannot Open File

The file specified could not be accessed or opened. (Mis-spelled?) The name of the file is output with the error message.

Filename Too Long

The filename specified was too long after the Librarian added a default extension.

Illegal Command

The command was not valid (mis-spelled?), or you used an ADD, DEL or REP command when no library was open.

Incompatible Object Module

You entered a library filename with the ADD command or an object module filename with the NEW command.

Input Line Too Long

The input line may not be longer than 80 characters.

Maximum Module Count Exceeded

You cannot have more than 256 modules.

Multiple Defined Global Symbol

A module being added to a library contains a global symbol name that exists in another module. The symbol name and the filename are output with the error message.

Multiple Defined Module

The name of a module being added to a library already exists in the library. Replace the old module or rename the new one. The module filename is output with the error message.

Must Be A Packed Object File

You used "all" or "ALL" with an ADD or REP command, but the file you specified was not a packed object file.

Multiple Defined Module

The name of a module being added to a library already exists in the library. Replace the old module or rename the new one. The module filename is output with the error message.

Not Enough Memory

There was not enough memory for the buffer or symbol table space required.

Read Error

An error occurred while a file was being read. The name of the file is output with the error message.

Seek Error

An error occurred while the position within a file was being updated.

Too Many Command Line Arguments

You invoked the Librarian with more than one system command line argument.

Undefined Module

The module being replaced or deleted from the library does not exist in the library.

Write Error

An error occurred while a file was being updated. The name of the file is output with the error message.

APPENDIX

Appendix A

Ascii Tables

Ascii Chart

Character	Binary	Octal	Decimal	Hex
NUL	00000000	000	000	00
SOH	00000001	001	001	01
STX	00000010	002	002	02
ETX	00000011	003	003	03
EOT	00000100	004	004	04
ENQ	00000101	005	005	05
ACK	00000110	006	006	06
BEL	00000111	007	007	07
BS	00001000	010	008	08
HT	00001001	011	009	09
LF	00001010	012	010	0A
VT	00001011	013	011	0B
FF	00001100	014	012	0C
CR	00001101	015	013	0D
SO	00001110	016	014	0E
SI	00001111	017	015	0F
DLE	00010000	020	016	10
DC1	00010001	021	017	11
DC2	00010010	022	018	12
DC3	00010011	023	019	13
DC4	00010100	024	020	14
NAK	00010101	025	021	15
SYN	00010110	026	022	16
ETB	00010111	027	023	17
CAN	00011000	030	024	18
EM	00011001	031	025	19
SUB	00011010	032	026	1A
ESC	00011011	033	027	1B
FS	00011100	034	028	1C
GS	00011101	035	029	1D
RS	00011110	036	030	1E

Character	Binary	Octal	Decimal	Hex
US	00011111	037	031	1F
SP	00100000	040	032	20
!	00100001	041	033	21
"	00100010	042	034	22
#	00100011	043	035	23
\$	00100100	044	036	24
%	00100101	045	037	25
&	00100110	046	038	26
'	00100111	047	039	27
(	00101000	050	040	28
)	00101001	051	041	29
*	00101010	052	042	2A
+	00101011	053	043	2B
,	00101100	054	044	2C
-	00101101	055	045	2D
.	00101110	056	046	2E
/	00101111	057	047	2F
0	00110000	060	048	30
1	00110001	061	049	31
2	00110010	062	050	32
3	00110011	063	051	33
4	00110100	064	052	34
5	00110101	065	053	35
6	00110110	066	054	36
7	00110111	067	055	37
8	00111000	070	056	38
9	00111001	071	057	39
:	00111010	072	058	3A
;	00111011	073	059	3B
<	00111100	074	060	3C
=	00111101	075	061	3D
>	00111110	076	062	3E
?	00111111	077	063	3F
@	01000000	100	064	40
A	01000001	101	065	41
B	01000010	102	066	42
C	01000011	103	067	43
D	01000100	104	068	44
E	01000101	105	069	45
F	01000110	106	070	46
G	01000111	107	071	47
H	01001000	110	072	48

Character	Binary	Octal	Decimal	Hex
I	01001001	111	073	49
J	01001010	112	074	4A
K	01001011	113	075	4B
L	01001100	114	076	4C
M	01001101	115	077	4D
N	01001110	116	078	4E
O	01001111	117	079	4F
P	01010000	120	080	50
Q	01010001	121	081	51
R	01010010	122	082	52
S	01010011	123	083	53
T	01010100	124	084	54
U	01010101	125	085	55
V	01010110	126	086	56
W	01010111	127	087	57
X	01011000	130	088	58
Y	01011001	131	089	59
Z	01011010	132	090	5A
[	01011011	133	091	5B
\	01011100	134	092	5C
]	01011101	135	093	5D
^	01011110	136	094	5E
_	01011111	137	095	5F
`	01100000	140	096	60
a	01100001	141	097	61
b	01100010	142	098	62
c	01100011	143	099	63
d	01100100	144	100	64
e	01100101	145	101	65
f	01100110	146	102	66
g	01100111	147	103	67
h	01101000	150	104	68
i	01101001	151	105	69
j	01101010	152	106	6A
k	01101011	153	107	6B
l	01101100	154	108	6C
m	01101101	155	109	6D
n	01101110	156	110	6E
o	01101111	157	111	6F
p	01110000	160	112	70
q	01110001	161	113	71
r	01110010	162	114	72

Character	Binary	Octal	Decimal	Hex
s	01110011	163	115	73
t	01110100	164	116	74
u	01110101	165	117	75
v	01110110	166	118	76
w	01110111	167	119	77
x	01111000	170	120	78
y	01111001	171	121	79
z	01111010	172	122	7A
{	01111011	173	123	7B
	01111100	174	124	7C
}	01111101	175	125	7D
~	01111110	176	126	7E
DEL	01111111	177	127	7F

Abbreviations for Ascii Control Characters

NUL	-	null or all zeros
SOH -		start of heading
STX	-	start of text
ETX	-	end of text
EOT	-	end of transmission
ENQ -		enquiry
ACK -		acknowledge
BEL	-	bell
BS	-	backspace
HT	-	horizontal tabulation
LF	-	line feed
VT	-	vertical tabulation
FF	-	form feed
CR	-	carriage return
SO	-	shift out
SI	-	shift in
DLE	-	data link escape
DC1	-	device control 1
DC2	-	device control 2
DC3	-	device control 3
DC4	-	device control 4
NAK -		negative acknowledge
SYN	-	synchronous idle
ETB	-	end of transmission block
CAN -		cancel
EM	-	end of medium
SUB -		substitute
ESC	-	escape
FS	-	file separator
GS	-	group separator
RS	-	record separator
US	-	unit separator
SP	-	space
DEL	-	delete

Appendix B

System Requirements

For **MSDOS** systems, the minimum system requirement is **640k** of available memory, minus the amount used by the operating system. Up to 30,000 line programs have been assembled with 640k of memory without running out of space.

The Linker assumes that **20 files** may be open at once. The **MSDOS** system default is 5. Your system disk contains the file **CONFIG.SYS**, in which you will see the line **FILES=XX**. You should change the number to at least 20. The Linker does not need more than 20 files to be open at any one time, since it opens and closes files to stay within the 20 limit. However, if any **memory resident programs** open files, you should add these to the Linker's 20.

UNIX systems require at least 1 megabyte of memory.

Index

A

Addressing Modes

Direct	1-12
Extended	1-13
Immediate	1-12
Indexed	1-12
Predefined Registers	1-14
Relative	1-13

Ascii Tables

Ascii Chart	A-2 - A-5
Ascii Control Character Abbreviations	A-2
	A-5

Assembler Directives

1-15 - 1-42

Assembler Directives - Assembly Mode

ABSOLUTE	1-25
COMMENT	1-27
ENDMOD	1-28
ENDS	1-25
INCLUDE	1-26
LEADZERO	1-26
MODULE	1-28
RADIX	1-26
RELATIVE	1-25
SECTION	1-24
SPACES ON	1-26
TWOCHAR OFF	1-27
TWOCHAR ON	1-27

Assembler Directives - Assembly Time

Absolute Versus Relative	1-41
Calculations	1-39
Comparisons	1-40
PAGE 0	1-40

Assembler Directives - Conditional Assembly

COND	1-29 - 1-33
ELSE	1-29
ENDC	1-32
ENDIF	1-32
EXIT	1-33
IF	1-29
IFABS	1-31
IFCLEAR	1-32
IFDEF	1-30
IFDIFF	1-30
IFEXT	1-31
IFFALSE	1-29
IFMA	1-32
IFNABS	1-31
IFNDEF	1-30
IFNDIFF	1-30

IFNEXT	1-31
IFNFALSE	1-29
IFNMA	1-32
IFNREL	1-31
IFNSAME	1-30
IFNTRUE	1-29
IFNZ	1-29
IFREL	1-31
IFSAME	1-30
IFTRUE	1-29
IFZ	1-29
Assembler Directives - Definition Control	
ARGCHK	1-21
ASK	1-23
DEFL	1-21
ENDM	1-22
EQU	1-21
EQUAL	1-21
EXTERN	1-23
EXTERNAL	1-23
GLOBAL	1-22
GLOBALS OFF	1-23
GLOBALS ON	1-23
LLCHAR	1-21
MACDELIM	1-22
MACEND	1-22
MACEXIT	1-22
MACFIRST	1-22
MACRO	1-21
MESSAGE	1-23
MESSG	1-23
PUBLIC	1-22
SET	1-21
VAR	1-21
XDEF	1-22
XREF	1-23
Assembler Directives - Linker Control	
COMREC	1-38
FILLCHAR	1-37
LINKLIST	1-37
OPTIONS	1-37
RECSIZE	1-37
SYMBOLS	1-37
Assembler Directives - Listing Control	
ASCLIST OFF	1-35
ASCLIST ON	1-35
CONDLIST OFF	1-34
CONDLIST ON	1-34
EJECT	1-36
HEADING	1-36
LIST	1-34
LIST OFF	1-34
LIST ON	1-34

MACLIST OFF	1-34
MACLIST ON	1-34
MLIST	1-34
MNLIST	1-34
NAM	1-36
NLIST	1-34
NOLIST	1-34
PAG	1-36
PAGE	1-36
PASS1 OFF	1-35
PASS1 ON	1-35
PL	1-35
PW	1-35
STTL	1-36
SUBTITLE	1-36
SUBTTL	1-36
TITLE	1-36
TOP	1-35
TTL	1-36
Assembler Directives - Macros	
Argument Separators	1-43
Examples	1-45
Labels	1-44
MACDELIM	1-47
MACEXIT	1-47
MACFIRST	1-46
Macro Definition	1-43
Mnemonic Definitions	1-45
Recursion	1-47
String Concatenation	1-44
Value Concatenation	1-44
Assembler Directives - Storage Control	
ASCII	1-18
BLKB	1-16
BLKL	1-17
BLKW	1-17
BYTE	1-15
C Specific - D_ALIGN	1-20
C Specific - F_ALIGN	1-20
C Specific - I_ALIGN	1-20
C Specific - L_ALIGN	1-20
C Specific - P_ALIGN	1-20
DATE	1-19
DB	1-15
DC	1-16
DEFB	1-15
DEFS	1-17
DEFW	1-16
DOUBLE	1-18
DS	1-17
DW	1-16
END	1-15
FCB	1-15

FCC	1-18
FDB	1-16
FLOAT	1-17
LONG	1-16
LONGW	1-16
LWORD	1-16
MASK	1-18
ORG	1-15
ORIGIN	1-15
RMB	1-17
SQUOTE	1-18
STRING	1-15
WORD	1-16
Assembler Error Messages	1-49
Assembly Language Syntax	1-9 - 1-11
High Byte	1-11
Labels	1-10
Local Labels	1-10
Low Byte	1-11
Number Base Designations	1-9
Program Comments	1-9
Program Counter	1-9
Upper/Lower Case	1-11
H	
High Byte	1-11
I	
Introduction	1-1
Addressing Modes	1-12 - 1-14
General	1-1
Major Version Number Differences	1-2
L	
Librarian	3-1 - 3-12
ADD, A	3-5
BOT	3-10
DEL, D	3-6
EXIT	3-6
FIND	3-7
General Description	3-1 - 3-2
HELP, H	3-7
Installation	3-3
Librarian Error Messages	3-11 - 3-12
LIST	3-8
NEW, N	3-8
Operating Instructions	3-4
QUIT	3-10
REP, R	3-9
STAT, S	3-10
TOP	3-10
Linker	2-1 - 2-19
Introduction	2-1
Linker Examples	2-7

Indirect Linking	2-15
Multiple Files with Multiple Sections	2-11
Single File Assembled at Desired Run Address	2-7
Single File with Multiple Sections	2-9
Single File with One Reference-Only Section	2-13
Linker Operating Instructions	2-2 - 2-6
Address Relocation	2-6
Command Line Mode	2-4
Data File Mode	2-3
Options	2-5
Prompt Mode	2-2
Linker Output Formats	2-20 - 2-24
2500 A.D. High Level Symbol Table	2-22
Abbreviated Global Symbol Table	2-21
Extended Intel Hex Format	2-26
Extended MicroTek Symbol Format	2-23
Global Symbol Table	2-20
Intel Hex Format	2-25
Motorola S19 Format	2-27
Motorola S28 Format	2-28
Motorola S37 Format	2-29
Zax Symbol Table	2-24

O

Operating Instructions	1-3 - 1-6
Command Line Mode	1-4 - 1-6
Error Only Listing to Disk	1-6
Error Only Listing to Printer	1-6
Error Only Listing to Terminal	1-5
Input and Output Filename	1-4
Input Filename Only	1-4
List On/Off to Disk	1-6
List On/Off to Printer	1-6
List On/Off to Terminal	1-6
Listing to a Different Drive or Directory	1-5
Listing to Disk	1-5
Listing to Disk with Cross Reference	1-5
Listing to Printer	1-4
Listing to Printer with Cross Reference	1-5
Listing to Terminal	1-4
Prompt Mode	1-3

S

System Defaults	1-7 - 1-8
Assembler Defaults	1-7
Assembler Error Processing	1-8
Assembler Run Time Commands	1-8
Assembler, Linker & Librarian	1-7 - 1-8
Librarian Defaults	1-7
Linker Defaults	1-7
System Requirements	A-1

25004DSOFTWAREINC.